Formalization Of Vertical Transformations In A Model Based Design Framework

THESIS

Submitted in partial fulfillment
of the requirements for the degree of
DOCTOR OF PHILOSOPHY
by

Anup Kumar Bhattacharjee
(Student ID: 2002PHXF406)

Under the Supervision of

Prof. R.K. Shyamasundar
School of Technology & Computer Science
Tata Institute of Fundamental Research

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE
PILANI (RAJASTHAN) INDIA
2008

**BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE
PILANI RAJASTHAN**

CERTIFICATE

This is to certify that the thesis entitled FORMALIZATION OF VERTICAL TRANSFORMATIONS IN A MODEL BASED DESIGN FRAMEWORK and submitted by ANUP KUMAR BHATTACHARJEE, ID.No.2002PHXF406 for award of Ph.D. Degree of the Institute, embodies original work done by him under my supervision.

Name: PROF. R.K. SHYAMASUNDAR
Designation: SR. PROFESSOR
SCHOOL OF TECHNOLOGY & COMPUTER SCIENCE
TATA INSTITUTE OF FUNDAMENTAL RESEARCH
MUMBAI

Date :

Acknowledgment

This research work would not have been possible without support from many people.

First of all, I would like to thank my supervisor, Prof. R. K. Shyamasundar for encouraging and guiding me throughout this work. I remember his first talk at a tutorial at TRDDC, Pune (in 1992) where he introduced me to the world of *synchrony* and I fell in love with *Esterel*. He is a very gifted and dedicated researcher. I am fortunate to have worked under him and I only wish that a small portion of his talents has touched me during our many meetings and discussions together. He has been always kind and patient with my shortcomings. I could never give my best as he expected.

Next to him, I would like to thank my superior colleague, Mr. S. D. Dhodapkar, Outstanding Scientist and Head, Software Reliability Section, BARC. He inducted me into the world of computer science after I joined BARC. It was his initiative that I first went into learning *lex* and *yacc* when my contemporaries were learning C. He is a man of remarkable capability of seeing things which most of us would have overlooked. I owe him for providing me the best support during my formative years in BARC. I wish to thank him for encouraging me to take up this research work and providing me with all organizational support.

I wish to thank BITS, Pilani for providing me financial support during this work without which this would not have been possible. In particular, I wish to thank Prof. V.S. Rao for encouraging me during my qualifying interview and later providing me support from BITS. I wish like to thank Prof Rahul Banerjee and and Prof Sundar Balasubramanian of Department of Computer Science & Engineering, for agreeing to be members of my Doctoral Advisory Committee. Their careful reading and comments have increased the quality of the thesis. I wish to thank Prof Ravi Prakash, Dean Research & Consultancy Division and Prof. Regalla Srinivasa Prakash, Asst. Dean, Research & Consultancy Division for their support from BITS. I also wish to thank Mr. Dinesh Kumar, Mr. Sharad Shrivastava and other staff members of Research & Consultancy Division who directly or indirectly assisted me during this period.

I wish to thank Mr. B.B. Biswas, Head Reactor Control Division, BARC, Mr. R.K. Patil, Associate Director (E&I Group), BARC and Mr. G.P. Srivastava, Director (E&I Group), BARC, for providing me the organizational support in BARC. I wish to thank Mr. R.K. Patil, Associate Director (E&I Group), BARC for taking personal interest in my research work. I also wish to thank Mr. G. Govindarajan ,Group Director , (E&I Group), BARC (now retired) for allowing me to take up the doctoral work.

I wish to thank Prof. S. Ramesh of IIT Bombay (now with GM R &D Lab, Bangalore) for constructive discussions during many of our collaborative projects and explaining me few finer aspects of Synchrony. I also wish to thank him for encouraging me to take up academic research work and register for Ph.D.

Thanks are also due to my colleagues Mr. Asif Iqbal (now with Honeywell) , Mr. Ajith K. John and Mr. Amol Wakankar who helped me with some of the implementations. It has been wonderful to work with all of them as they made the lab environment lively.

I wish to thank my parents for providing me with good education and making me a good human being. They always gave me the best that they could afford.

I wish to thank my parent-in-laws for supporting me. They are wonderful people who sacrificed many things for us and supported me during all my difficult days of parenthood.

I also wish to thank my elder brother (Dada) who did lot of hand holding during my school days. He taught me Physics and Mathematics during my formative years and initiated me into the world of scientific learning.

My son Arijit deserves lot of thanks for being a wonderful son. I am grateful to him for having patience with me when I did not spend the time that I should have spent with him. He kept us in good humour with $\forall$ and $\exists$.

Lastly, no word of appreciation is good enough for my wife Ruby. She supported me during most difficult days and kept me in good spirit. She is a brave lady who could face many daunting task and has a spirit that never gives up. I cannot repay my dues when she suffered the most during our early days of parenthood while I was away at IIT Kharagpur.

Abstract

Model-based design methods emphasize concurrency, communication abstractions, and temporal aspects, rather than only procedural interfaces. A successful model based design is a methodology based on mathematical and visual methods and addresses the problems associated with designing complex information and control systems. The model-based design paradigm is significantly different from the traditional design methodology. Rather than using complex structures and extensive software code, designers can now define advanced functional characteristics using building blocks which are defined in terms of primitive functions having precise meanings. These built models along with appropriate simulation, automatic code generation and verification tools can lead to rapid prototyping, software testing and verification.

In this thesis, we address few issues in a model based design framework concerning enrichment of modeling languages, transformation of the modeling language to a framework which eases verification and code generation, detection of run-time execution errors and validation of translation from high level language to low level implementation.

The key areas of focus in this thesis are:

- Formalization of translation of the graphical (UML oriented) notations typically used in in model based designs.
- A language to model choreography of service oriented computing for distributed systems.
- Type system of weakly typed language like C to detect run time errors.
- A scheme for translation validation to validate the assembly code produced by the compiler.

We show formalization of two graphical notations namely Statecharts and Activity Diagrams which are used in model based design methodologies. Statechart is a class of hierarchal statemachine and is one of the modeling language used for specifying the reactive behaviour of an entity based on it's response to events. Statecharts are commonly used in the specification and design of embedded systems. However no formal treatment of the semantics of Statecharts has been given as a part of the OMG standard. Without a formal semantics, specifications in Statecharts are not amenable to analysis and automatic code generation.

Activity Diagram used in UML and STATEMATE encapsulates activities performed by the system and is used to show graphically a process view of a system in terms of interactions between processes. It can be used to depict the high level processing view of the system which is a composition of distributed components, each performing local activities.

We show how these visual notations can be realized in a synchronous language framework. This underlying synchronous semantics allows verification of the models created in

these notations. It also allows high level imperative style code generation using compilers for synchronous languages. Since the model can be verified and code is generated from the verified model using provably correct transformation, our approach addresses the concerns of correct-by-construction approach in vertical model transformation.

The traditional semantics proposed in OMG standards lacks features to model reactive behaviour of workflows. It also needs semantic enrichment when each of the activities is prone to failure and need compensating actions. Such extensions have wide applications in modeling and debugging complex business processes. We have proposed extension to Activity Diagrams to model failure and compensations.

Service oriented computing is a new emerging paradigm for distributed computing. Services are autonomous computational entities and web services are the most common application of service oriented computing. The terms *orchestration* and *choreography* are used to define two different flavors of service oriented computing. While orchestration is used to describe a single view point model, choreography is about specifying the service orchestration in a global model. Choreographs define the sequence of exchanging messages between two (or more) independent participants or processes by describing how they should cooperate. We focus on structuring choreography as a set of message exchanges among various participating roles as conversations. In this thesis, we also define a language framework, which integrates orchestration with scripting to abstract conversations leading to an effective modular specification of service choreography.

We address the inadequacy of a type system in a language like C popularly used in embedded system design as a part of the back end tool chain. We show a method of defining a type system of C in a deductive framework. The method is based on a novel model of C programs: each C program is modeled as a typed transition system encoded in the specification language accepted by PVS theorem prover. Since the specification is strongly typed, proof obligations are generated, for possible type violations in each statement in C, when loaded in the PVS theorem prover which need to be discharged. The technique does not require execution of the program to be analysed and is capable of detecting typical type errors such as array bound errors, divide by zero, arithmetic overflows and underflows etc.

We describe a methodology and a system for the validation of translation of a simple HLL to assembly language programs. Our method consists of converting the high level language (HLL) program and its object code to a common semantic representation such as Fair Transition System (FTS), and then establishing that the object code is a refinement of the HLL program. We show that the the proof of refinement can be performed using a theorem prover. In our examples, we have used Stanford Temporal Prover (STeP) as the theorem prover. The proposed approach also has the additional advantage that the embedded system remains unaffected by compiler revisions/updates.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Designing software for safety-critical systems such as those used in aerospace, control of nuclear reactors, medical systems, large distributed web applications etc., is a complex and challenging task. This requires a systematic approach involving modeling, analysis and simulation techniques as have been used beneficially in other branches of engineering. This approach supported by verification is expected to achieve improvements in dependability of software designs.

Model-based architecture and design [91] methodology is increasingly becoming popular in software used in safety-critical applications. A model enables to analyse the properties of a proposed design solution for a system before actually realising it in software and/or hardware. The methodology is based on the representation, composition, and manipulation of models during the design process. An emerging common requirement in model-based software and systems design is that modeling languages need to be domain-specific (Domain Specific Modeling Language (DSML)), offering software/system developers abstractions and notations that are tailored to characteristics of their application domain.

Models once created should be amenable to rigorous analysis (model analysis) to check that their behaviour meet the actual system requirements. It is also essential to be able to generate efficient implementation (model based code generation) of the system from the model. Model analysis and model-based code generation require the precise semantics of modeling language. One of the most commonly used modeling language in software specification and design is Unified Modeling Language (UML) [92]. The Unified Modeling Language (UML) is a general-purpose visual modeling language that is designed to specify, visualize, construct and document the artifacts of a software system. UML is used in specifying software systems from embedded applications to business process applications. Considerable effort is required to ensure the correctness of the specification in terms of simulation and testing. However simulation and testing only show the desired behaviour but cannot prove the absence of an undesirable behaviour. Approaches based on formal verification techniques have been found to be successful in handling such requirements and it is being increasingly used (particularly in hardware)to find corner cases (difficult bugs) as a complementary effort to testing. It is required to have a rigorous semantics behind such graphical notations which allows rigorous verification of the models. Additionally, if the actual implementation of the specification is not derived from the model on which verification is done, the implementation will have potential to diverge from the specification. Hence there is a requirement of automatic code generation from the models through a process of model transformation.

Model based design typically works as a phase wise model transformation and refinement. The transformation works at two levels: Vertical and Horizontal. Vertical transformation is typically synthesis of a new artifact from a description at a more abstract level. Horizontal transformation is more like refactoring an implementation for efficiency or portability. Continuous verification in these phases needs to be a part of the model based design, which ensures correctness-by-construction [52]. The verification requirements are with respect to

1. Correctness of the model with respect to certain system invariants.
2. Correctness of mapping model to a high level language implementation.
3. Type correctness of the implementation.
4. Correctness of the translation to a machine language.

These are most demanded by software developments in safety-critical systems. As demands for safety-critical software increase, future requirements will necessitate strategies to support these in a tool driven environment and hence needs to be addressed as part of the vertical transformations in a model based design environment.

While UML is rapidly becoming the industry standard for modeling, its standard definition does not contain a precise semantics which helps in model verification and code generation.

The main objective of this thesis is to provide a formal interpretation of some of the graphical notations typically used in UML, within a framework of synchronous methods and process calculus. This will address requirement of precise specification of UML models and can support continuous verification. It will allow formal model analysis and model-based code generation in a vertical model transformation framework. We show formalization of two graphical notations namely Statecharts [56, 57] and Activity Diagrams [92] which are used in model based design methodologies. Statecharts and Activity Diagrams are extensively used in UML notations, although the actual syntax used in UML are slight variation of the syntax used for this thesis. We also discuss a formal language based on a reactive framework for describing choreography in distributed service oriented systems.

As a part of the vertical transformation, the high level model is progressively transformed into a High Level Programming Language (HLL) like C which can be efficiently compiled into a machine executable binary. Such model generated code should be *type safe*. This is particularly true if automatic code generation from models is employed in safety-critical systems. One of the issues with types arise because of arithmetic with finite precision. The inability of computers to represent an infinite range of values is well known. For example, in the case of signed or unsigned integers, it is useful to define a valid range, within which all values are guaranteed to lie after the result of an assignment or initialization on that integer type. It is necessary to define a type system policy to be enforced in the event that a resulting assignment or initialization to variables through evaluation of expressions lie inside the valid range.

It is important to ensure that the High Level Language (HLL) code generated from the model is translated correctly to a machine language by a compiler. In several key safety-critical embedded applications, it has become mandatory to verify the process of translation by compilers since usually compilers are only certified rather than verified. We have explored the application of Translation Validation [98] in the verification of compiled code against the input source program.

This thesis, in addition to the formalization of the Statecharts, Activity Diagrams, also addresses the type correctness issues of the high level code from the model and correctness of translation to machine language implementation.

1.2 Related Work

Several domain specific notations and methods have been developed to help the designer specify clear and unambiguous system requirements, verify that the requirements are consistent and correct, and verify that the refined design meets its specification. Notable among these are Unified Modeling Language(UML) [92], Systems Modeling Language (SysML) [69], Simulink/Matlab [61] and Business Process Modeling Notation(BPMN) [93]. Most of the notations like that of UML, which are used in industry are based on graphical syntax. However many of these notations do not have a rigorous semantics because of which the specifications created using these cannot be subjected to rigorous analysis. The advantage of such analysis notably results in proving some of the required design properties of safety and liveness [29]. Automatic 100% code generation from the modeling language is also not possible if the modeling language lacks a precise semantics. Lack of semantics also leads to model and code going out of sync, if there is a change in requirement or a change in manually developed code from the model.

The area of research in semantics of visual languages for state based model driven design has been very active during the last decade [76, 57, 4, 55, 36]. Andre [4] has presented a formal semantics of hierarchical statemachines in a synchronous framework. However the presentation does not include the complete syntax of Harel's Statecharts [57]. Formal semantics of Statecharts to that presented here covers the complete syntax (restricted to input deterministic class) of Statecharts.

The suitability of activity diagrams for modeling business process has been argued in

[109]. To the best of our knowledge, the first formal semantics of UML Activity Diagram (UML AD) was proposed by Eshuis [41, 42]. Eshuis proposes the semantics at the following two levels :*Requirement Level* and *Implementation Level*. The first level is based on Statechart like semantics and is transformed into a transition system for model checking by NuSMV [30]. The second level is based on STATEMATE [54] semantics of Statecharts extended with properties to handle data. The semantics covers activity charts of UML 1.5 but not of activity diagrams of UML 2.0[1]. A token flow semantics based on Petri Nets was proposed in [106] by mapping activities into *procedural Petri nets*, which excludes data type annotations but includes control flow. Storrle [105] has defined mappings to *procedural Petri nets* to prevent multiple calls which otherwise would result in infinite nets. However these approaches do not address the automatic code generation as may be required in a tool driven environment. Semantics based on synchronous language was proposed in [15] which allows a validated code generation from the notation.

A process algebraic formulation of workflow is proposed in [117]. A theoretical foundation of flow composition languages is given by Bruni in [24]. Fu [48], presented an approach of converting BPEL Web services into guarded statement and further into PROMELA/SPIN for verification. Similar approaches based on Finite State Processes (FSP) [80] and CCS were presented in [46] and [74]. However they didn't consider failure and subsequent compensation and hence not suitable for modeling business processes. Business Process Execution Language (BPEL) support the notion of compensation in case of a service failure. An activity can be associated with another activity that acts as its compensation action. This compensation handler can be invoked either explicitly or by the default compensation handler of the enclosing scope. In this sense, UML Activity Diagrams need to be enriched with additional semantic constructs to model such compensating actions. The extensions of Compensating Activity Diagrams proposed in this thesis is inspired by Hoare [107] and is based on a flow composition language introduced in [25]. The notion of compensation was perhaps first introduced as *Sagas* in handling Long Running Transactions by Garcia-Molina and Salem [49]. A formal model for compensable transactions has been defined by Li [78]. Verification approaches for incorporating compensating transactions were reported in [5, 40, 45].

Orchestration and choreography are emerging standards for creating business processes from multiple Web services. Solutions are required to have a good modeling support to specify such applications. The idea of Scripting as an abstract language for modeling conversation based on communication patterns was presented in [47]. Research in these area is very active in terms of orchestration languages based on a process calculus model [73, 86, 114] and verification issues [38, 40].

Another area of focus in this thesis was to formulate a formal language to capture the global choreography in distributed service oriented computing. WS-CDL [111] is the first choreography language proposed by W3C in 2005. WSCI [110] is another language targeted toward building a global interconnection model linking service operations but is not actually a chore-

[1]It should be pointed out that UML 2.0 is a significantly re-engineered version of UML 1.5, particularly in the context of activity diagrams.

ography language. An excellent theoretical framework for service choreography is discussed in [99]. Orchestration is the prime language abstraction supported by various programming languages such as BPEL [64], BPML [90]. Another elegant formalism referred to as Orc [86] has been proposed for orchestration. Conformance validation between Choreography and Orchestration has been reported in [77]. As pointed out in [86], a reactive semantics for web services will provide a good basis for static and dynamic resource discovery and exploitation as it leads to good dynamic monitoring schemes. The later is also one of the aims of our study and proposal.

We also focus our attention to type correctness and correctness of translation of high level code via model transformation. Most of the model based design tools produce High Level Language (HLL) programs e.g., in C Language. It is well known that C has weak type system and cannot guarantee absence of *Run Time Error(RTE)*. Static Analysis tools like SPLint [43] are very useful in detecting RTEs in C, related to buffer overflow and pointer arithmetic but they do not address RTEs associated with arithmetic expressions. The program supervision tools e.g. those built using Valgrind [89] require target program execution and suffer from the same limitations as that of testing. Semantic Analysis based on Abstract Interpretation [33] is a more rigorous approach to detect runtime errors statically. This technique is based on data flow analysis which computes program properties by converting programs into equations over data types represented as lattices and then solving these equations over these lattices. Tools such as PolySpace [62], ASTREÉ [32] use the abstract interpretation technique for the detection of runtime errors in C programs. However these tools work on an abstract model (sound but not complete) of a program and are prone to *false positives* which are possible errors requiring further effort to investigate. Commercial tools also do not allow tuning to reduce false positives. There are also excellent research works on software model checking like BLAST [58], SLAM [8] etc., which are used for checking specifications related to software interfaces, shared resources in device drivers etc. These tools model check specified property and do not specifically address the issues of arithmetic runtime errors in a global manner.

It has remained a grand challenge to establish the correctness of a compiler [59]. For establishing the correctness of a compiler, one has to prove that the compiler always produces target code that correctly implements the source code. Owing to the intrinsic complexities of compiler verification, an alternative referred to as *Translation validation* has been explored in [98]. In this approach, each individual translation (i.e. a run of the compiler) is followed by a validation phase which verifies that the target code produced on this run correctly implements the source program. Such a possibility is particularly relevant for embedded systems where there is a need to execute a finite set of target programs. It must be pointed out that the validation task becomes increasingly difficult with the increase of sophistication and optimizations methods like scheduling of instructions as in RISC architectures or methods of code generation/optimization for super-scalar machines[88]. In [98], the authors demonstrated the the practicability of translation validation for a translator/compiler that translates the synchronous language Signal to C without any optimizations[1]. We have demonstrated in [14] the application of translation validation in validating translations of High Level Language (HLL)

[1]In [97], extension of the approach for TNI Signal compiler is explored.

compilers like C and Ada for a restricted class of programs.

1.3 Benefits of the Approach

Model-based design methods emphasize concurrency, communication abstractions, and temporal properties, rather than only procedural interfaces. A successful model based design is a methodology based on mathematical and visual methods and addresses the problems associated with designing complex information and control systems. The model-based design paradigm is significantly different from the traditional design methodology. Rather than using complex structures and extensive software code, designers can now define advanced functional characteristics using building blocks which are defined in terms of primitive functions having precise meanings. These built models along with some simulation tools can lead to rapid prototyping, software testing and verification. For example, domain-specific models for embedded systems might represent physical processes using ordinary differential equations [61], signal processing using dataflow models [72, 50], and decision logic using finite-state machines (FSM)[[12]. Model-based design methods address system specification, model transformation, synthesis of implementations, model analysis ,validation, execution, and design evolution. Model based design methodology is now also being sought and advocated for large distributed software development for managing business processes.

Model-based design of software uses formal, composable and manipulable models in the design, implementation and system integration process. Modeling languages introduce layers of abstractions in the design flow that are synergistic with the design objectives and the nature of the system to be designed. This is partly achieved by meta-modeling languages and meta-models describing the abstract syntax (concepts, relationships and well-formedness rules) of such modeling languages.

Synchronous language framework [51, 11, 75] is a sound methodology to address embedded software and hardware design, validation, and implementation. The advantage lies in rigorous mathematical semantics of synchronous languages, which allows one to argue about the consistency and correctness of the model and it's correct realization as a compilable implementation. The overall idea is to generate correct-by-construction [52, 53], deterministic implementation from high-level rigorous specifications. The advantage is in increasing quality while decreasing design and validation costs. In the correct-by-construction approach the process of software development is treated as in other branches of engineering, and are based on development of mathematical models, design, verification and refinement. A system is deterministic, if it always reacts in same way to the same inputs occurring with the same timing. On the contrary, a nondeterministic system can react in different ways to the same inputs, actual reaction depending on internal choices or computation timings. It is known that determinism is a must in safety critical system software.

A sound methodology is most sought after in the development of control applications like aerospace, nuclear, chemical, automotive and robotic applications. It is also being increasingly used in development of software used in business processes involving transactions. It provides an efficient approach for the five key elements of the development process cycle ("V" diagram): Requirements capturing, Specification, Development, and Deployment of the system, thus integrating all these phases and providing a common framework for communication throughout the entire design process. To support this, it is beneficial to have a sound framework to model, verify and derive the implementation seamlessly directly from the specification.

1.4 Contribution of the Thesis

The main contribution of the thesis are highlighted below:

1. Formalization of semantics of Statecharts based on an imperative synchronous semantics of ESTEREL : We present algorithms [18] [RESS01] for the translation of Statecharts into Esterel. This allows us to use other backend tools in the synchronous family for verification and high level code generation for implementation .

2. Extending the Statecharts to model communication: We present an extension in Statecharts to model communication through channels. The new language allows us to model communicating reactive systems. We also present an alternate translation [66] [SAFECOMP2003] scheme to Promela used in the verification tool SPIN. We also show how such a specification can be realized [19] [IT2008] in the imperative synchronous language Esterel.

3. Formalization of Activity Diagrams in Esterel: We present a reactive semantics of the various activity patterns using Esterel. We show that this semantics allows us to carry verification and code generation using tools used for family of synchronous languages [15] [ICDCIT2005].

4. Compensating Activity Diagrams: Although the Activity Diagrams can model most of the workflow patterns used in business process, however it cannot model failures in business processes. We show a possible extension of activity diagrams to model compensations required in such business process logic [JOT].

5. Language to Model Choreography in Distributed Service Oriented Computing: Choreographs define the sequence of exchanging messages (conversations) between two (or more) independent participants or processes by describing how they should cooperate. In [16, 17], we have explored the use of Scripts in describing conversational aspect of choreography. We show a formal language framework which integrates orchestration with scripting to abstract conversations leading to an effective modular specification of service choreography [16], [17] [APSCC08,ICWS08].

6. Type systems for C: Most of the model based design tools generate the implementation in C language. This is particularly true for tools used in embedded systems. However it is

known that there is no proper type system for C and hence C programs may not be type-safe (e.g. array index overflow, arithmetic overflows). We show a deductive technique based on PVS for verifying type safety of C progarm. We also show an implementation for checking type safety for a restricted class of C programs based on a type system implementation in PVS [67][SAFECOMP07].

7. Object Code Validation: The code generated from a model using a model compiler needs to be ultimately translated to a machine level code using a HLL compiler. It is known that compiler verification is an undecidable problem. We show how an alternate methodology based on Translation Validation can be used to verify fragments of High Level Language programs e.g C and it's correct translation to an object code (assembly instructions) [14][FTRTFT00].

1.5 Outline of the Thesis

The thesis is presented as follows: It is divided into two major parts: Part I describes the visual notations, their semantics and translation algorithms and Part II describes the work related to type safety of model generated High Level Language (HLL) and translation validation for a HLL compiler.

Part I is organized as follows: Chapter 2 introduces the general definition of reactive systems, Statecharts as a notation to model reactive systems and its semantics. The formalisation of translation of Statecharts in a synchronous language framework of ESTEREL is presented in Chapter 3. An extension of communication in Statecharts to model communicating reactive processes(CRP) is presented in Chapter 4. Chapter 5 presents a formalization of Activity Diagrams in a process algebraic framework and introduces an extension of Activity Diagrams to model compensations in business process logic. An implementation of Activity Diagrams is presented in Chapter 6. Chapter 7 presents the language framework to define service choreography.

Part II is organized as follows: In chapter 8, a methodology to build a type system for C programs is presented. In chapter 9, we present a methodology for compiler validation based on the framework of Translation Validation.

Part I

Visual Modeling Notations for Reactive & Workflow Systems: Formalization and Realization

Chapter 2

Statecharts : Visual Modeling Notation for Reactive Systems

2.1 Introduction

Reactive systems are computer systems that react continuously to their environment, at a speed determined by the latter. This class of systems contrasts, on one hand with transformational systems (classical programs whose inputs are available at the beginning of their execution and which deliver their outputs when terminating: for instance compilers), and on the other hand with interactive systems (which react continuously to their environment but at their own speed: for instance operating systems). Among reactive systems are most of the industrial real-time systems (control, supervision, and signal-processing systems), as well as man-machine interfaces. These systems have the main following characteristic:

- Parallelism. The design must take into account the parallelism between the system and its environment. It is convenient and natural to design such systems as sets of parallel components that cooperate to achieve the intended behavior.
- Determinism. These systems always react in the same way to the same inputs. This property makes their design, analysis, and debugging easier. Thus, it should be preserved by the implementation.
- Temporal requirements. These requirements concern both the input rate and the input/output response time. They are induced by the environment and must imperatively be matched. Hence, they must be expressed in the specifications, they must be taken into account during the design, and their satisfaction must be checked on the implementation.
- Reliability. This is perhaps their most important feature as these systems are often critical ones. For instance, the consequences of a software error in an aircraft automatic

pilot or in a nuclear plant controller are disastrous. Therefore, these systems require rigorous design methods as well as formal verification of their behavior.

A language well suited to the specification and design of reactive systems should, therefore, allow specification of parallel and deterministic behaviours and allow formal behavioural and temporal verification.

2.2 The Synchronous Approach

Synchronous languages have been introduced in the 80s to make the programming of reactive systems easier [12]. The purpose of these languages is to give the designer ideal time primitives, thus reducing the chance of programming misconceptions. Instead of the interleaving paradigm, they are based on the simultaneity principle: All parallel activities share the same discrete time scale. Concretely, this means that $a \parallel b$ is viewed where a and b are simultaneous. Each activity can then be dated on the discrete time scale; this has the following advantages:

- Temporal reasoning is made easier.
- Interleaving based nondeterminism disappears, which makes program debugging, testing, and validating easier.

Concerning the implementation, the idea is to project this discrete time scale onto physical time. As the scale is discrete, nothing occurs between two consecutive instants: Everything must happen as if the processor running the program were infinitely fast. This is the outcome of *synchrony hypothesis* which can be stated as

1. the system evolves through an infinite sequence of successive atomic reactions indexed by a global logical clock,
2. during a reaction each component computes new events for all its output signals based on the presence/absence of events computed in the previous reaction and,
3. the communication of events among components occur instantaneously between two successive reactions.

Although such an infinitely fast processor does not exist, but it suffices that any input be treated before the next one. In order to verify this condition, one only needs to know the maximal input frequency, and an upper bound on the execution time of the object program. For this purpose, synchronous languages have deliberately restricted themselves to programs that can be compiled into a finite deterministic interpreted automaton, a control structure whose transitions are deterministic sequential programs operating on a finite memory. Each transition, whose execution time is statically computable, corresponds to the system reaction to an input. There are numerous languages based upon the synchrony hypothesis: ESTEREL [12], LUSTRE [50], SIGNAL [9], STATECHARTS [56], etc. Significant advantages of the family of synchronous languages include the availability of idealized primitives for concurrency, communication and preemption, a clean rigorous semantics, a powerful programming environment with the capability of formal verification. The advantages of these languages are nicely paraphrased by Gerard Berry, the inventor of ESTEREL , as follows: *What you prove is what you execute*.

Textual and graphical formalisms have their own intrinsic merits and demerits. For instance consider the following reactive specification of control flow (switching of tasks) among various computing tasks and interrupt service tasks in a control software. The computing tasks switch from one to another in cyclic fashion and are shown as substates of *compute_proc*. The interrupt service tasks are entered as a result of the occurrence of interrupt events. The history notation has been used to indicate that on return from interrupt tasks, the system returns to last executing compute task (except when event *100ms* occurs, the control returns to compute task *hpt*). Such systems can be specified using graphical formalisms easily. The Statechart

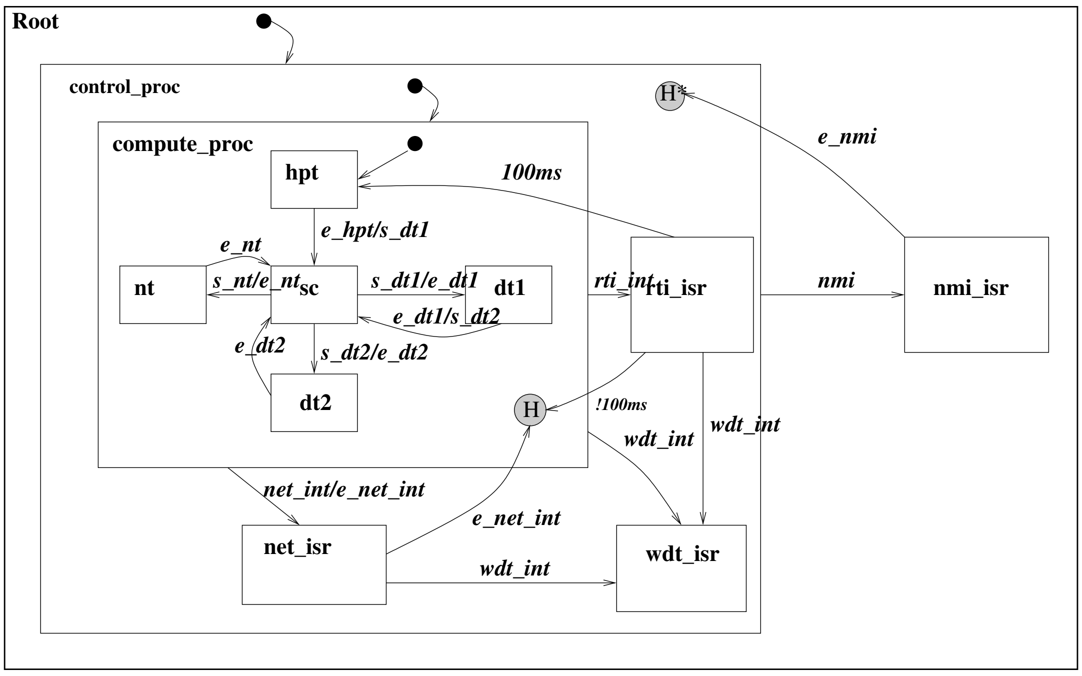


Figure 2.1: An Example Specification of a Real Time System

for the above system is shown in Figure 2.1. Lets us consider the intended behaviour: *When the event wdt_int occurs on system failure, the state of the system will be in state wdt_isr and subsequently the system will toggle between states wdt_isr and nmi_isr.* Arguing the correctness of this intended behaviour from such descriptions, however, is not easy. Our work is concerned with methods that will combine advantages of using graphical formalisms for the design of reactive systems with that of using formal verification tools in textual formalisms.

Statecharts is a visual formalism which can be seen as a generalization of the conventional finite automata to include features such as hierarchy, orthogonality and broadcast communication between system components. Being a formalism rather than a language, there is no unique semantics in the various implementations and further Statechart specifications can be nondeterministic. For these reasons, even though there are powerful programming environments for Statecharts such as STATEMATE (which includes simulators), environments lack formal verification tools. The primary motivation incorporating the Statecharts-to-Esterel translator tool was to open up possibilities of formal verification which seemed easily possible via ESTEREL route, the code generation/optimization capability coming as a bonus. In fact, the implementation of the tool has been found to have the important advantage of using Statecharts or ESTEREL for the specifications/modeling of different components of the same system.

2.2.1 A Brief Introduction to ESTEREL

For a better understanding of the synchrony hypothesis, let us study some examples in ESTEREL . ESTEREL is an imperative synchronous programming language. Besides variables, the language manipulates signals: A signal can be valued or pure, and can be an input signal (its presence can be tested), an output signal (it can be emitted), or a local signal (it can be emitted and its presence can be tested). The communication mechanism is the synchronous broadcast: any signal emitted by someone at a given instant is received by everybody at the same instant. Moreover, the temporal primitives of ESTEREL are intuitive, which will make the following examples easy to understand: Since control is passed instantly from a finishing statement to the next one, the statement await 5 Second; await 5 Second is equivalent to await 10 Second.1 For the same reason, in the statement

```
every 60 MINUTE do
   emit HOUR;
end every
```

the signal HOUR is simultaneous with the 60th occurrence of the signal MINUTE. There is no notion of physical time inside a synchronous program, but rather an order relationship between events (simultaneity and precedence). The physical time is thus an external signal, like any other external signal. As a result, one can write either `abort TRAIN when 10 METER` or `abort TRAIN when 5 SECOND`. In the statement

```
present A then
   % something
end present;
||
present B then
   % something else
end present
```

each component of the parallel construct can react independently to its signal. As a consequence, the program reacts either to A alone, B alone, or A and B at the same time.

These small examples show that the synchrony hypothesis leads to very natural code. Providing the designer with ideal temporal primitives greatly reduces the number of programming errors. The drawback is that, once compiled, the execution time of the program must match the temporal specifications. But of course the same problem arises with an asynchronous programming language like ADA. Finally, it is important to note that the synchronous approach has been validated through several real-life projects.

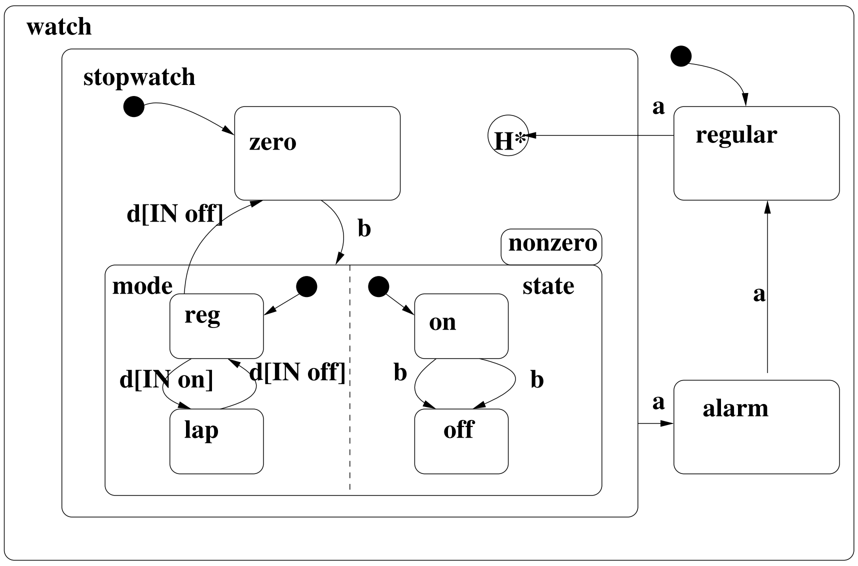


Figure 2.2: Statechart Specification of a Stopwatch

2.3 Components of a Statechart

States: There are three types of states; *Basic state*, *Or-state* and *And-state*. Basic states are those states which do not contain any other state, eg., *lap* is a basic state.

An *Or-state* is a compound state containing two or more other states. To be in an *Or-state* is to be in one of its component states. In this presentation, we will use *Or-State* synonymously with XOR-state, i.e., we can be in only one of the component states at a given time. An example of an *Or-state* in Figure 2.2 is *stopwatch*.

An *And-state* is also a compound state and staying in an *And-state* implies staying in each one of its substances. This is provided to model concurrency. The substates of an *And-state* may contain transitions which may be executed simultaneously. The state *nonzero* shown in Figure 2.2 is an *And-state*.

Transitions: A Transition in the Statechart is a five-tuple (source, target, event, action, condition). The arrow on the Statechart goes from source to target and is labeled as `e[C]/a`, meaning that event e triggered the transition when condition C was valid and action a was carried out when the transition was taken. In general, a could be a list of actions to be taken.

History and Defaults: Statechart incorporates the idea of a history state in an OR-State. The history state keeps track of the substate most recently visited. This is denoted by H in a Or-state, as in the or-state stopwatch in Figure 2.2. A default state, marked by a shaded circle, is a substate of an or-state such that if a transition is made to the or-state and no other condition (e.g. enter-by-history) is specified, then that substate must be entered by default, e.g. *regular* is the default substate for the watch. In Figure 2.2 , we have a deep-history state, which means that a transition to that state implies being in the maximal most recent set of basic substates. This can be represented by history states in each one of the Or-substates.

2.4 STATEMATE Semantics

The informal semantics of the STATEMATE version of Statecharts is provided through rules describing the semantics of a step. The main rules are listed below. For detailed discussions, the reader is referred to [57].

1. Reactions to external/internal events and changes that occur in a step can be sensed only after completion of the step.
2. Events are "live" for the duration of the step following the one in which they occur only.
3. Calculations in a step are based on the situation at the beginning of the step.
4. If two transitions are in conflict, then priority is given to that transition whose scope is higher in the hierarchy. The scope as defined in [57] is: The scope of a transition tr is the lowest Or-state in the hierarchy of states that is a proper common ancestor of all sources or targets of tr, including non-basic states that are explicit sources or targets of transition arrows appearing in tr.
5. Each step follows the Basic Step Algorithm as described in [57].

2.5 Formalization of STATEMATE Semantics of Statecharts

Finite State machines (FSM) are widely used in the modeling in various areas. They are used to represent the flow of control and are amenable to formal analysis based on model checking technique.

Definition 2.5.1 *A FSM consists of*

- *a finite set of Q of states,*
- *a finite alphabet Σ,*
- *an initial state $q^I \in Q$,*
- *a final state $q^F \in Q$,*
- *a set $\rightarrow \subseteq Q \times \Sigma \times Q$ of transitions,*

Given a word $\rho = \sigma_0\sigma_1 \cdots \sigma_n$ over the alphabet Σ, an accepting run of the FSM M over ρ is sequence

$$q_0 \xrightarrow{\sigma_0} q_1 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_n} q_{n+1} \tag{2.1}$$

such that q_0 equals the initial state q^I and q_{n+1} equals to the final state q^F and for $0 \leq i \leq n, (q_i, \sigma_i, q_{i+1}0$ is a transition of M. The set of words $\rho \in \Sigma^*$ over which M has an accepting run is called the language of M, denoted $L(M)$.

FSMs can be composed parallelly and hierarchically to build Hierarchical Machines(HM) [2] called Statecharts(SC). It is assumed that these FSMs have mutually disjoint set of states.

Definition 2.5.2 *We define a Transition System as a special FSM which has no final state. A Transition System is a triple S=($Q, q^I, \rightarrow$), where S is a set of states, $s \in S$ is the initial state, $\rightarrow \subseteq Q \times L \times Q$ is the set of transitions. A reactive system can be defined as the composition of such transition systems. If S is a reactive system and S produces output O at the moment that input I is provided, the behaviour can be written as*

$$S \xrightarrow[I]{O} S'$$

Definition 2.5.3 *A HM is defined inductively as*

- *Base case: A TS $(Q, q^I, \rightarrow)$ is a HM.*
- *Concurrency: If $M_1, M_2 \cdots M_k$ are SCs then $M_1 \parallel M_2 \parallel \cdots \parallel M_k$ is a HM.*
- *Hierarchy: If $\mathcal{M}$ is a finite set of HMs, $\gamma = (Q, q_I, \rightarrow)$ is a TM with states Q and $\mu : Q \mapsto \mathcal{M}$ that associates each state $q \in Q$ with a HM in $\mathcal{M}$ then the triple $(\gamma, \mathcal{M}, \mu)$ is a HM. Here γ is defined to be the root of the SC.*

A HM of the form $M_1 \parallel M_2 \parallel M_k$ is called a product expression and each M_i is called a component of the product expression. Such product forms are called AND-Charts.

Definition 2.5.4 *We define a HM of the form $(\gamma, \mathcal{M}, E, \mu)$ as a Statechart, where the TS γ is called the top-level of the hierarchical expression, and each HM in $\mathcal{M}$ is called a component of the Statechart. E is defined as the set of events which are propositions over Σ and their boolean combinations.*

The set of events are defined over the alphabet Σ as boolean predicates

- $e \in \Sigma \to e \in E$.
- $e \in \Sigma \to \neg e \in E$
- $e_1 \in \Sigma$ and $e_2 \in \Sigma \to e_1 \wedge e_2 \in E$
- $e_1 \in \Sigma$ and $e_2 \in \Sigma \to e_1 \vee e_2 \in E$

The mapping function μ defines the type of Statecharts.

- If $\mid \mu(s) \mid= 1$ then s is defined to be a single flat TS.
- If $\mid \mu(s) \mid=> 1$ then s is defined to be a parallel composition of TS.
- If $\mid \mu(s) \mid= 0$ then s is defined to be a basic state.

The composition function μ on M defines a successor function such that

$$\chi : \bigcup_{M \in \mathcal{M}} Q_M \to \mathbb{P}(\bigcup_{M \in \mathcal{M}} Q_M)$$

defined by

$$\chi(s) = \{s' \mid \exists M \in \mathcal{M} \bullet M \in \mu(s) \wedge s' \in Q_M\}$$

where $s' \in \chi(s) \leftrightarrow s'$ is a state of the direct sub-automaton of s. We define a configuration as the set of states in which the system lies at any instant of time.

Definition 2.5.5 *Given a Statechart $SC = (\gamma, \mathcal{M}, E, \mu)$. A set $C \subseteq \bigcup_{M \in \mathcal{M}} Q_M$ is a configuration of SC iff*

- *There exists one state of the root automaton $\gamma \in C$*
- $s \in C \wedge M \in \mu(s) \to \exists i \mid s_i \in Q_M \to s_i \in C$
- $s \in C \wedge (\exists s' \bullet s \in \chi(s') \to s' \in C)$

A transition $s \xrightarrow{l} s'$ of $M \in \mathcal{M}$ is enabled if $s \in C$ and $(C, E) \models l$. Let $Enabled_{C,E}$ be the set of enabled transitions. An automaton $M \in \mathcal{M}$ is said to be enabled if $s \xrightarrow{l} s' \in Enabled_{C,E}$. We define two functions source and target such that $source(s \xrightarrow{l} s') = s$ and $target(s \xrightarrow{l} s') = s'$. If t1 and t2 are two transitions then t1 is at a higher priority if $source(t1) \prec source(t2)$, where $\prec$ is the preceding operator.

Definition 2.5.6 *Harel [57] defines the execution of a Statechart as an execution step in (C,E) which consists of synchronously firing all the transitions in a maximal non-conflicting set of transitions. If $\Upsilon \subseteq Enabled_{C,E}$ be the set of enabled transitions. Υ will be maximal and non-conflicting iff*

- $\forall t \in \Upsilon \; \nexists t' \in Enabled_{C,E} \bullet Source(t) = Source(t')$
- $\forall t \in Enabled_{C,E} \bullet t \in \Upsilon \leftrightarrow \nexists t' \in Enabled_{C,E}$

2.5.1 Synchronous STEP Semantics

If two reactive systems S_1 and S_2 make a step together, they are defined to be concurrent and the global step represented as $S_1 \parallel S_2 \xrightarrow[I]{O} {S_1}' \parallel {S_2}'$ which may be composed of two local steps $S_1 \xrightarrow[I]{O} {S_1}'$ and $S_2 \xrightarrow[I]{O} {S_2}'$.

Definition 2.5.7 *Let $K = (S, s_0, \overset{STEP}{\rightarrow})$ be TS where*

- *$S = Conf(\mu) \times \mathbb{P}(E)$ is the set of states of K, where $Conf(\mu)$ is the set of all configurations.*
- *$s_0 \in S$, where $s_0 = (C_0, \phi)$ is the initial state of K where C_0 is defined as follows. Let $\gamma_{root} = (Q, q^I, rightarrow)$ and $S_0 = \bigcup_{M \in \mathcal{M}} s_{0_M}$ then $C_0 = (\chi \mid Q_0)^*(s_0)$*
- *$\overset{STEP}{\rightarrow} \subseteq S \times S$ is the transition relation of K where $(C, E) \overset{STEP}{\rightarrow} (C', E')$ such that $C' = (C/\ \chi^*(Source(\Upsilon))) \cup Target(\Upsilon) \cup (\chi^+(target(\Upsilon)) \cap \bigcup_{M \in \mathcal{M}} s_{0_M})$ and $E' = E \cup Action(label(\Upsilon))$*

This definition of composition models synchronous behaviour. This gives us following advantages

1. The reaction time is short as possible and we take it as 0.
2. The timing behaviour is abstract enabling us further refinement without bothering about delays since $0 + 0 = 0$

The execution semantics of a SC now can be defined in terms of steps of execution of the composite systems. Responsiveness, Modularity and Causality [63] are three very important criteria in judging the semantics.

Definition 2.5.8 **Responsive** *A semantics S is responsive if for any two distinct input sets I_1 and I_2 and non-empty output set O with $O \cap (I_1 \cup I_2) = \phi$ such that*

$$S \xrightarrow[I_1]{O} \text{ and } S \not\xrightarrow[I_2]{O}$$

Definition 2.5.9 Modularity *A semantics is modular if for any two systems S_1 and S_2 the following two statements are equivalent*

1. $S_1 \parallel S_2 \xrightarrow[I_1]{O} S'_1 \parallel S'_2$
2. $S_i \xrightarrow[I \cup O_{i-1}]{O_i} S_i'$ *for i=1,2*

where $O = O_1 \cup O_2$

Definition 2.5.10 Causality *A semantics is causal if we can add to every step $S \xrightarrow[I]{O} S'$ a partial order $\leq$ on $I \cup O$, such that*

1. *if $S \xrightarrow[I]{O}$ and $S \not\xrightarrow[I']{O}$ and $I, O \neq \phi$, then there is at least one dependency between I and O i.e $\exists a \in I, b \in O$ with $a \leq b$*
2. *if $S_1 \parallel S_2 \xrightarrow[I_1]{O} S'_1 \parallel S'_2$ with causal order $\leq$ then there should exist a partitioning into processes $T_1, T_2 \cdots T_n$ and causal orders $\leq_1, \cdots \leq_n$ such that $\leq \restriction (I_i \cup O_i) = \leq_i$ and $T_1 \parallel \cdots \parallel T_n = S_1 \parallel S_2, n \geq 2$, and for each $T_i \xrightarrow[I_i]{O_i} T_i'$, these steps combine into the steps of $S_1 \parallel S_2$*

It has been proved in [63], that no semantics of reactive systems can be responsive, modular and causal at the same time. In our framework, we assume the following

- Every observable step is divided into a number of micro-steps. Actions and reactions strictly follow the order in micro-steps. The events generated as a reaction to some input can only be sensed in the micro-step following the input.
- Events are generated at the next step but before the reaction of the system is completely stable (no more transitions are possible), no input from environment is possible.

We will explain later that this is achieved in our translation of Statecharts into ESTEREL by dividing the micro-steps in ESTEREL using an external signal *STEP*. The introduction of the *STEP* signal makes the semantics nonmodular but responsive and obeys ESTEREL causality [12]rules. In general, Statecharts allow specification of nondeterministic behaviour. It also allows non causal behaviour. However in creating specifications of reactive systems, we need to have systems that are output deterministic in a nondeterministic environment. ESTEREL is a Turing complete imperative language whose semantics is defined based on Meije synchronous process calculus. It only allows to specify deterministic output behaviour in the presence of nondeterministic inputs. It is possible to represent a restricted class (deterministic & causal) of Statecharts into ESTEREL . This enables one to model check and verify Statecharts using the ESTEREL environment. The other advantage is that such a representation enables one to generate high level validated code for final implementation.

2.6 Summary

In this chapter, we have discussed the visual notation of Statecharts and its semantics as proposed by Harel [57]. We have discussed few example Statecharts typically used to specify the behaviour of a real time system. We have also discussed a brief formal interpretation of the *STEP* semantics.

Chapter 3

Realization of Statecharts in the Textual Formalism of ESTEREL

3.1 Introduction

In this chapter, we describe a method of translating Statechart formalisms into ESTEREL with the idea that the powerful verification tools and code optimization tools of ESTEREL can be applied for Statechart programs. Our aim has been to provide a clean formally verifiable code for Statechart programs rather than yet another attempt to define the semantics of Statecharts. For this reason, we stick to using the STATEMATE semantics [57], which is an industrial strength version of Statecharts. It must be noted that ESTEREL is deterministic and hence, our study is confined to the deterministic class of Statecharts. However, it may be noted that the translation procedure will detect the underlying nondeterminism if any.

The translation preserves the correspondence between the Statechart states and the ESTEREL program components for aiding debugging of the programs and verification of the properties at different levels.

3.2 Structure of the Translator

The internal stages of the translator are schematically shown in Fig.3.1. The first stage is the preprocessing tool **stpp**. The **stpp** module converts syntactical constructs like join points, conditional points etc. into a simpler but equivalent notation. These constructs are designed to provide a crisper notation while drawing Statecharts but can be simplified to simple transitions. This helps in simplifying the implementation of translator which can then expect fewer constructs in the input. It also checks for violation of Statechart syntax and gives warning messages.

The second stage of the translator is the functional form generator **stffg**. The output of

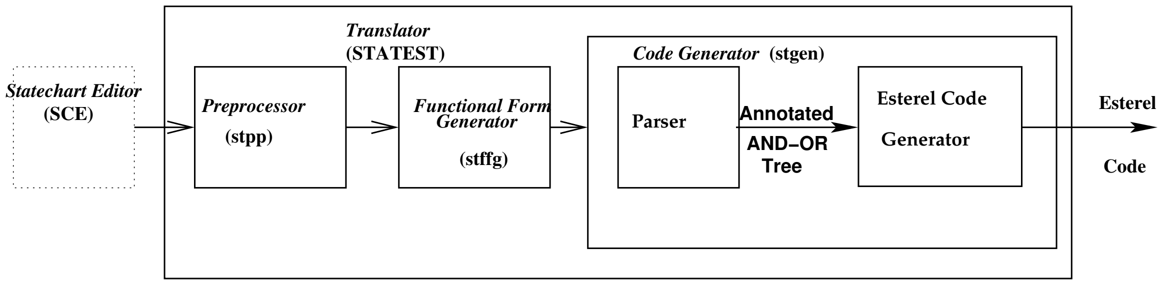


Figure 3.1: Schematic Diagram of the Translator

stpp is fed into the functional form generator **stffg**, which converts the graphical notation into a textual form for further translation. This textual form is described by a context free grammar and hence amenable to rigorous syntax checking and translation. The third stage **stgen** is the ESTEREL code generator. It takes the functional form as input and emits ESTEREL code. The textual representation of the Statecharts in LALR(1) form is given in appendix A.

3.3 Semantic Transformation of Statecharts into ESTEREL

In this section and subsequently in the chapter, we describe the basic algorithmic requirements for the transformation of Statecharts into ESTEREL . The actual algorithm is however described in the next chapter. Any transformation of Statecharts must preserve the following

- The hierarchy of states and transitions,
- Conflict resolution in the transitions as per the STATEMATE semantics
- Transitions between states,and,
- Support of communication via events and actions.

In the following, we shall highlight the underlying issues of representation, resolution of conflicts and code generation. Note that we refer to signals in the Statechart as actions or events, while those in ESTEREL are referred to simply as signals. We first present the underlying ideas and the actual code generation algorithm is presented at the end.

3.3.1 AND-OR Tree Representation of Statecharts

The Statechart can be represented as an AND-OR tree: being in an AND-node meaning that the system is in each of its child nodes. Such a representation allows us to express the hierarchy of states of the Statecharts in a convenient manner to trace the path of arbitrary transitions.

This also allows us to resolve conflicts between enabled transitions, by calculating the scope (refer to section 2.2). For purposes of code generation, we actually use an annotated representation of AND-OR tree described in the following section. An AND-OR tree representation of the Statechart is shown in Figure 3.2.

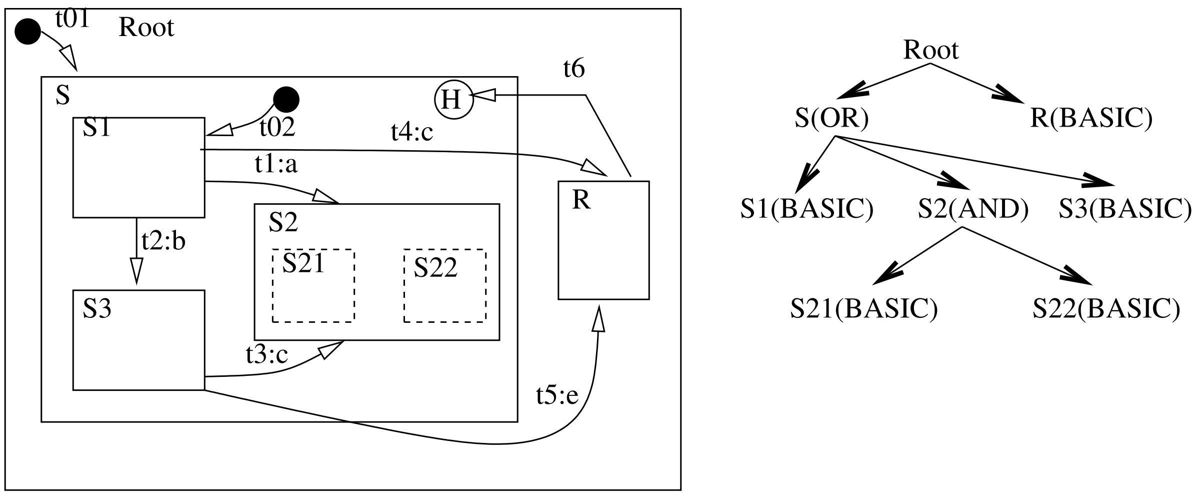


Figure 3.2: Statechart and its AndOr Tree Representation

The annotated AND-OR tree keeps track of information about the Statechart pertinent for the translation, such as

1. the states and their types,
2. hierarchy of States, and
3. Transitions between states which includes Entry and Exit points for each transition that exists a state.

Each node A of the AND-OR tree is represented as a seven-tuple[1]:

$(Name, Type, T_{entry}, T_{exit}, T_{loop}, T_{default}, T_{history})$, where,

- Name : Name of the state viz. S.
- Type : AND, OR or BASIC.
- T_{entry} : The set of all transitions that enter S (e.g., for state S2, T_{entry}=$\{t_1, t_3\}$).
- T_{exit} : The set of all transitions that exit S (e.g., for state S1, T_{exit}=$\{t_1, t_2, t_4\}$).
- T_{loop} : The set of all transitions that exit one of S's immediate child states and enters another (possibly same) child state (e.g., for state S, T_{loop}=$\{t_1, t_2, t_3\}$).

[1]We shall use node synonymously with state and vice-versa

- $T_{default}$: The single transition to an immediate child state from A (e.g., for state S, $T_{default}$=$\{t_{02}\}$).
- $T_{history}$: The set of transitions to the history state of A (e.g., for state S, $T_{history}$=$\{t_6\}$).

We need to keep track of the Entry and Exit Point Information so that the transitions including the inter-level transitions can be enabled in the translated ESTEREL code preserving the STATEMATE semantics. The actual information we need to keep track will be clear by considering the states between which the transition takes place. Transitions in Statecharts can be broadly classified as:

- T_1 : Between child states of the same parent (e.g.,Transitions $\{t_1, t_2, t_3\}$ in Fig.3.2).
- T_2 : From a parent state to its (not necessarily immediate) child state(e.g.,Transitions $\{t_{01}, t_{02}\}$ in Fig.3.2) .
- T_3 : From a child state to (not necessarily immediate) parent state.
- T_4 : Any transition that is not of Type T_1,T_2 or T_3 (e.g., $\{t_4\}$ in Fig.3.2).

Note that all of these transitions may not occur in a given Statechart. In particular, types T2 and T3 may not occur, but the way they are translated forms part of the translation for type T4. Thus for example in Fig. 3.2, the transition labelled t_4 would make T_3 types in states subordinate to `S` and T_2 in state `Root`. The book keeping of the above classes of transitions is achieved through the Node- Labeling Algorithm by keeping the appropriate entry and exit information in each node and the AND-OR tree.

3.3.2 Node-Labeling Algorithm

Node-Labeling Algorithm: Assuming levels of the nodes in the tree have already been computed (with root node having level 0, and increasing level for its child nodes), for each transition in the set Tr of transitions, the algorithm traverses the path from source node n1 to target node n2, labeling these two nodes as well as intermediate nodes with:

1. name of the transition,
2. type of the transition, viz., T_1,T_2,T_3 and T_4 and
3. the fact whether the transition is entering that node or exiting it.

This information is used to generate code in the translation.

3.3.3 Labeling for Transition Conflict Resolution

As per STATEMATE semantics, two transitions are in conflict if they exit a common state A. Further, conflict resolution is based on the following: Transition t_1 has priority over transition t_2 as defined earlier(if the lowest[2] Or-state exited by t_1 is lower than the lowest Or-state exited by t_2). Given this, if trigger events for t_1 and t_2 occur simultaneously then, we must ensure that t_2 is not taken along with its actions. This is done by a signal $hide\_A$. On taking t_1, $hide\_A$ will be emitted. Therefore, before t_2 is taken, a check must be made for the presence of signal $hide\_A$. This is indicated in the AND-OR tree by traversing the tree top-down, maintaining a list of "hide signals" that we need to label the nodes with. At a node, which has at least one transition that exits it, we label all of its children with $hide\_A$. This is to ensure that while translating, a statement to check for the presence of $hide\_A$ is executed before any transition is taken. This will perform the job of hiding internal signals. The algorithm to implement $hide\_signal$ labeling is omitted here for brevity.

3.3.4 Structure of the ESTEREL Code

We illustrate the structure of the ESTEREL model for each type of states from Statechart. Each state in Statechart is modeled as a module in ESTEREL .

Transformation of a Basic state: If A is Basic-state, then the ESTEREL code generated for state A has the form

```
module A :
           emit EnterA;
           do sustain InA watching ExitA;
end module
```

Both `EnterA` and `InA` are global signals. The first statement broadcasts an internal event `EnterA` of entering the state A. The `do sustain InA watching` statement simulates the effect of being in state A by ensuring that while the state A is occupied the `InA` signal is emitted every instant and is available for other states for testing if required. The `do .. watching ExitA` construct ensures that as soon as the signal `ExitA` is emitted, the statement `sustain InA` , terminates in the same instant (strong preemption)

Transformation of a Super-state: While translating a superstate of a Statechart, the following has to be implemented.

1. Entry to this superstate

[2]Lowest means closest to the root node

2. Emission of signals to cause entry to substates. The substates may be entered as direct destinations or as a history or as defaults, if the first two are absent.

3. All transitions exiting the substates. Transitions of type 4 are broken into multiple transitions of type 2(child to parent) and type 3(parent to child), happening together in one step.

4. Transitions of loop-type w.r.t this superstate.

Transformation of an Or-State : If S is an OR-state with substates, say, S1, S2,S3 . . . Sn, then the structure of the ESTEREL module has the following structure.

```
module S :
        Block_1 || Block_2 || Block_3
end module
```

The || symbol represents parallel execution of three code blocks in ESTEREL syntax.

`Block_1`: This block in the beginning contains statements broadcasting two signals `EnterS` and `InS` signifying entry to superstate S. A local signal ("go" prefixed to the name of the immediate substate to be entered) is then emitted to cause entry to one of the substates. The signal is captured in `Block_2`.

`Block_2`: This block of ESTEREL code contains n parallel segments, n being equal to the number of immediate substates, one segment corresponding to each of the substates. This is of this form

```
            Code_for_S1 || Code_for_S2 || ..  || Code_for_Sn
```

Each of the segments contain ESTEREL statements for entering the module of each substate as well as code for all transitions exiting that substate.

`Block_3`: This block of the module handles all inter-level transitions which are of Loop-type w.r.t. this state. This transition is similar to type 1 transition (sibling to sibling) except that the transition originates from deep within one sibling and may terminate deep within another sibling. During a transition this superstate which is the common ancestor of the source and destination state of the transition receives a valued signal from immediate child state exited due to the transition and emits a signal causing entry to the destination state. The emitted signal is simple "go" if the destination is immediate child. Otherwise entry is caused to the ancestor of the destination state, by emitting a valued signal prefixed with "sig_D", the value being the numerical identifier of the destination state. All of the interposing superstates will relay this signal to their substates till the final destination state is reached.

Transformation of And-state: If S is an And-state with substates S1, S2, . . . Sn , the structure of the ESTEREL code has the following form:

```
module S :
         Block_1 || Block_2
end module
```

In And-state the code segment `Block_1` is different from the corresponding `Block_1` code segment of an Or-state, in that, there is no emission of local *go* signal which causes a substate to be entered as in Or-state. The reason is when an And-state is entered all of its substates are entered at the same instant by definition. There is no `Block_3` because there cannot be Loop-type transitions for And-states. The `Block_2` code has n parallel segments (n= no of substates) but there is no *await immediate go..* construct as there is no corresponding emission of a go signal in `Block_1`.

Handling Nondeterminism

Some enabled but conflicting transitions cannot be resolved by scope rule [57]. For example, there can be two enabled transitions leaving the same state having the same scope. This type of nondeterminism is permitted in Statechart and is expected to be handled by simulation environment either by randomly taking one transition or allowing the user to decide about which transition to take. Since ESTEREL is a purely deterministic language, the translation scheme has to prioritize the transitions. We resolve this by generating the *await case .. end* construct. Each case statement corresponds to an event triggering the transition, whereby whichever *case* is positioned first is executed when more than two conflicting events occur.

3.3.5 ESTEREL Code Generator

The ESTEREL code generator is the most complex stage of the translator and its internal structure is shown in the Figure 3.1. Here we present a brief description but the details along with all translation algorithms can be found in [18, 13, 103]. It consists of a parser which parses the input Statechart (in the functional form) and constructs an annotated AND-OR tree by a syntax directed translation scheme. This AND-OR tree is the input to the code constructor.

Translating a "state" in Statechart into ESTEREL means generating a code segment in ESTEREL which will represent the occupied state in Statechart. Such a code segment in ESTEREL , representing an occupied state, has to repeatedly perform ("sustain") the actions expected be carried out in that state, while waiting for events which will cause that code segment to be terminated and other code segment (a new state) to be entered. Such code segments linked together by ESTEREL statements which watch for the events in the system and pass on control from one code segment to other (as per the specified transitions in Statechart) constitute the full translation of Statechart program to ESTEREL program. In ESTEREL code, the transitions are implemented by *await* statements. In principle it is possible to construct a single module in ESTEREL representing the entire Statechart behaviour. However such a monolithic ESTEREL code is difficult to understand and also does not permit visualization of occupied/unoccupied states when simulated using ESTEREL simulator *Xes*. In the code generation strategy described here, one ESTEREL module is generated for each state in the input Statechart. This results in the generation of modular ESTEREL code capturing the behaviour of the Statechart. The only disadvantage of this scheme is that the code generated contains large number of global signals which are seen on the simulator panel.

Figure 3.2 shows a Statechart and its AND-OR tree alongside. Each node of the AND-OR tree represents the corresponding state in the Statechart and is annotated with the information like name, type, set of all transitions which enter and exit this state, history flag indicating whether history exists for this state etc.

The annotated And-Or tree is the central data structure for the code generation phase. The code generation algorithm traverses the tree in reverse postorder visiting all the children nodes from left-to-right before visiting the parent node. While visiting each node, the ESTEREL module for each node (State in Statechart) is emitted and the signals required to interface various modules are collected. This list is pushed up the tree for the purpose of global signal declarations in each related module. In the following, we explain the structure of ESTEREL code generated for different types of states in Statechart.

The translation is done in a top down manner traversing the AND-OR tree. In short, the process is as follows:

1. Declare all necessary signals,
2. Generate code for states and transitions between states,
3. Generate code to communication within the Statechart,
4. Generate code to deal with special constructs such as history substates.

Declarations: Information about the following kinds of signals is stored in the annotated AND-OR tree and these are declared at each node while generating code for the module corresponding to that node:

1. External Input signals.
2. Internal Input events generated during transitions out of substates of parent node A.
3. Internal Output events (actions) generated during transitions out of substates of parent node A.
4. If A is a substate of an Or-state with history, then a valued signal new_history_A is used to that the history can be changed appropriately whenever transition to a substate A_i of A takes place.

5. Dummy signals for T_2 or T_4 transitions that enter A: In this case signals of the form sig_BtoA or sig_AtoB would be needed, where B is either an immediate parent or an immediate child of A. This list is built up for each such node A, during Node Labelling Algorithm. These signals are used to build a chain of signals that trigger transitions between immediate parent-child states, and the whole chain generates the entire transition.
6. Dummy signals for T_3 and T_4 transitions that exit A. This is similar to 5 above.
7. Valued History signals for all Or-sub-states having history; for each such OR-state these store the value of the most recent substate. While building the AND-OR tree we can maintain a list of Or-states which have history.
8. Signals that indicate transition to a history substate of a substate of A, or if A is an Or-state, to indicate transition to history substate of A.
9. Characteristic signals (in, enter, exit) for each substate of A. To generate this list, traverse the AND-OR tree bottom-up (postorder) and at each node, add to a list of child nodes. Then while generating code for node A, declare all characteristic signals for each of its child nodes as listed.

We have a new module only for each OR-node, therefore, we need not keep a list of all nine types of signals with an AND-node or BASIC-node unless it is the ROOT node.

The STEP signal: In the ESTEREL code generated, each step occurs on receipt of an external signal called STEP. This signal is needed to provide a tick on which transitions can be made even when there are no input signals from the environment (i.e., when all triggering events are internally generated). Use of STEP is necessary to implement the super-step semantics of STATEMATE, wherein several steps are executed starting with some initial triggering events, till the system reaches a set of stable states (i.e., states with no enabled transitions out of them).

Transitions: Consider code generation for the translation for a transition t of Type T, with source state A and target state B. In brief, the model transformation involves the following:

- Generate code to await the occurrence of the triggering event, and
- On occurrence of the STEP (as in STATEMATE semantics), if the triggering condition is true and no transition preempts t, emit:

  [-] a signal to activate the next state (called a "go" signal),

  [-] a signal to activate a chain of transitions (for types T2 through T4),

  [-] signals to exit the current state, i.e., to terminate emission of signals that depict the current state as active.

Figure 3.3 illustrates transformations with respect to T_4 transition. In the ESTEREL code for state A, we only show the code for state C and the transition labelled *a/b*.

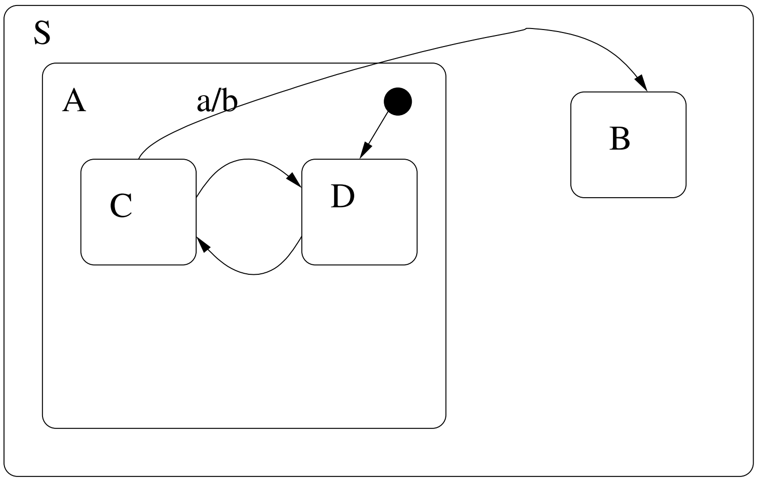


Figure 3.3: Translation of T4 Type Transition

```
module S
   loop
      [
         await immediate goA ;
         trap TA in
            [
               run A ;
               exit TA
            ]
         end trap
         present sig_AtoS then
            emit goB ;
         end present
      ]
      loop
         [
            await immediate goB ;
            trap TB in
               [
                  run B
               ]
            end trap
         ]
      end loop
   end module
```

Code for Module S

```
module A
   trap TA in
      [
         loop
            [
               await immediate goC ;
               trap TC in
                  [
                     run C ;
                      ||
                     await immediate a ;
                     await STEP ;
                     emit b ;
                     emit sig_CtoA ;
                     exit TC
                  ]
               end trap;
               present sig_CtoA then
                  [
                     emit sig_AtoS ;
                     exit TA
                  ]
               end present
            ]
         end loop
      ]
   end trap
]
end module
```

Code for Module A

3.4 Code-Generation Algorithm

In the following, we describe the basic model transformation algorithm. Code to be emitted for immediate states like history and special actions are omitted for brevity.

Notation: In the code-generation algorithms, the emitted code is shown in EMIT blocks.

3.4.1 Algorithm:GenCode

The main algorithmic steps are briefly described below:

A1 Traverse the AND-OR tree top-down. (in preorder)
For each node A do

A2 If A is an OR-node:

(a) Begin a new module, and declare all signals that occur A's signal list, or in the signal list of child nodes of A, till the first child Or-node is encountered.

(b) Generate code for each block representing the substate of A. Let A_1, A_2,. . ., A_n be the immediate child nodes of node A. Let $e_{i1}, e_{i2}, \ldots, e_{im}$ be the external or internal events on which transitions are made out of the A_i. Let the corresponding actions be act_{i1} to act_{im}. Further, let the conditions under which the transitions are to be taken be C_{i1} to C_{im}. Let the list of hide signals for the nodes A_i, $\forall i$ be $hide_1$ to $hide_t$. *STEP* is a signal that indicates that the next step must be performed. It is an external signal. Steps of the translation are described below:

(c) **Step 1** Emit preamble code. If A is a substate of an OR-state B with history, then appropriate *newhist* signals are emitted to update history. Code to be emitted from this step is given below:

```
emit enter_A;
[trap $T_A$ in [
       sustain In_A;
       || await tick ; emitnewhistB(A);]
       ||[ signal $goA_1, goA_2, ..goA_n$ in [
```

(d) **Step 2** Emit code to check for T_2 and T_4 transitions, or for transitions to the history substate of A. If none of these are true then default state is entered. Code from this step is given below:

```
present
       case $sig\_AtoA_j$ do
              emit $goA_j$
       case entehist_A do
                     [ if histA = 1 then
                          emit $goA_1$
                     elseif histA=2 then
                          emit $goA_2$
```

```
                              else emit $goA_k$
                              endif
                       end case
              end present
```

(e) **Step 3** For each i, emit code to handle transitions out of A_i and also the refinement of A_i. The code for each of the i are composed in parallel. The respective codes to be emitted are given in the substeps below:

Substep 1. Preamble code for Ai.

```
[ loop [
       await immediate $goA_i$;
       trap $T_{A_i}$ in ...
```

Substep 2. Emit code corresponding to the refinement of A_i. We indicate the refinement of A_i by $<< A_i >>$. If A_i is an AND-node or BASIC-node then this is the block of A_i. If A_i is an Or-node, then this is a `run` A_i statement. In this case, add it to a queue of Or-node Q_{nodes}, so that we emit code for it and its child nodes later. When the node is visited during the preorder tree traversal, the entire subtree under that node A_i is skipped to be processed later.

```
[ « $A_i$ » ;
       exit $T_A$;
       || ....
%subsequent codes will be completed in other steps
```

Substep 3. Emit code for each transition triggered by $e_{i,j}, j = i..m$, and composed in parallel with the above code, i.e.,$\forall t_i \in {T^i}_{exit}$.

```
call translateTransition(t,$A_i$);
end trap %$T_A$
```

Substep 4. Code emitted in case there are transitions of type T_3 or T_4. Thus for all transitions t of type T_3 or T_4 which exit state A_i, we would have:

```
       call exitCodeTrans(t,$A_i$);
```

Substep 5. Postamble code for the substate Ai is given below:

```
] end loop
end
```

(f) **Step 4.** The postamble code to be emitted is given below:

```
] end signal
end
```

A3 If A is an AND-node:

Generate code to emit enter and in signals for A, or for updating history, as in preamble code above.

Generate code for each one of A's child nodes, A_i, and compose these in parallel

Generate code for each transition that quits a child node of A and compose each in parallel with that in item 2 above. The translation for the individual transitions is exactly as for an Or-node. There are no looping transitions of type T_4 and AND-node.

A4 If A is a BASIC-node:

Generate code to emit enter and in signals for A, or for updating history of its parent state, just as was done for the Or-state. Also generate code to begin, await a return signal for or end an activity.

1. Generate code for each of the Or-nodes in the queue Q_{nodes} till no more Or-nodes remain in the queue.

3.4.2 Algorithm : TranslateTransition

This procedure gives the translation for a translation t of type T with source state A and target state B.

```
procedure translateTransition(t:Transition,currNode:Node)
begin
    A:=source(t);
    B:=target(t);
    $e_t$ := event(t);
    $a_t$ := action(t);
    $C_t$ := condition(t);
/* Let hideS be signal which hides all transitions of scope less than t*/
    if (A = currNode) then
    begin
        EMIT[
        loop
            await immediate $e_t$;
            await STEP;
            if $C_t$ then
            [
                present $hide_1$ else
                    present $hide_2$ else
                    ...
                        present $hide_n$ else
                            emit hideS;
                    emit $a_t$;
                    emit exit$_A$;
                  end..
      ]       if t.trType= T1 then
      begin
         EMIT[
            emit go$_B$;
            emit exitT$_A$;
      end
      ]       if t.trType= T2 OR t.trType=T4 then
      begin
      $\forall S_i \in ancestor(A)$ and $child(S_i, B)$
         EMIT[
            emit sig_AtoS$_i$;
            emit T$_A$;
      end
      ]   else
   /* A $\neq$currNode*/
   begin
      EMIT
      [          present sig_AtoA$_1$ then
            emit sig_A$_1$toA$_2$;
            end
   end    ] end procedure
```

3.4.3 Generation of *STEP* signal

In the above Algorithm described in 3.4, each step occurs on receipt of an artificially created external signal called *STEP*. Clearly, this *STEP* signal cannot be generated internally, as it will not generate a tick then. Further, *STEP* must be given to the state machine (system) as long as there are enabled transitions (enabled on internally generated signals). In our translation, this indication is obtained from the enter and exit signals emitted. We define a new signal *"give_step"* which is emitted whenever an enter or exit signal is emitted. Thus, whenever give_step is emitted, a *STEP* signal must be emitted. Additionally, *STEP* must be emitted on occurrence of an external input. The state machine generated by the ESTEREL compiler must interface with the environment through a driver routine. The driver routine executes the state machine whenever there is an input from the external environment. Thus, our problem is to execute the state machine under certain conditions (namely when *give_step* is emitted) even when there is no external input. The trick here is to set a bit for every occurrence of *give_step* that is checked by the driver routine; the bit indicates that the driver routine must generate a tick (and supply a *STEP*). Thus, due to the presence of *"await STEP"* in the translation for transitions, although the actions are "activated" in the current step, they take effect only in the next step. This is in accordance with the STATEMATE semantics.

Our model faithfully represents all behaviours of the STATEMATE Statecharts, in both the *Step* and *Superstep* time models. In our translation, the STEP of Statecharts is mapped to the tick of ESTEREL. Time instants are indicated by a separate TIME signal. In the *Superstep* time model, the STEP and TIME signals are distinct, while in the Step model they always occur together. As noted in [57], a Statechart using the *Superstep* time model can have possible infinite loops in the same TIME instant. This can also happen in our ESTEREL translation, and cannot be detected using the present ESTEREL tools.

Let us consider the Statechart shown in fig.3.4. Following are the steps executed when the event a occurs.

- STEP 1: Transition tl is enabled because of occurrence of a and the system moves from the configuration R,A to R,B and the event b is generated in the system.
- STEP 2: In this step since event b is available, transition t2 is enabled and the system leaves the configuration R, B and moves to R, C and the event c is generated.
- STEP 3: In this step since event c is available, transition t3 is enabled which when taken leaves the configuration R,C, moves on to R,A and a is generated.

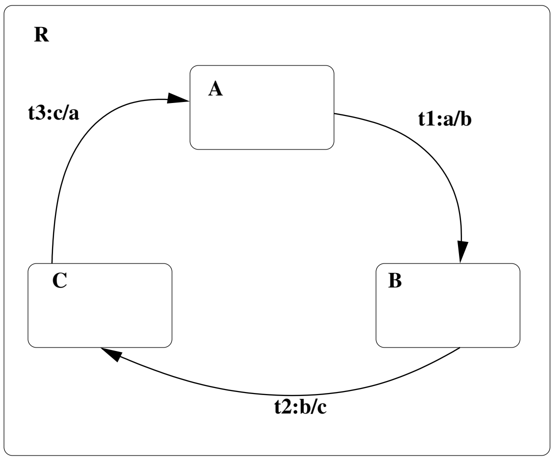


Figure 3.4: Cyclic Transitions

In the asynchronous time model [57], all these steps will constitute one superstep and be executed in one time instant. Each of these steps is executed when the external signal STEP is given.

3.5 History

As noted in [57], history states can occur only within Or-states. History is implemented using valued history signals for each Or-state having history. The value 0 corresponds to the default state, i.e., no history. The emission of the history signals for a state S, histS is done only by the root module ROOT, of the entire Statechart. When a new state is entered within an Or-state S, the module corresponding to that state emits a *newhistS* signal which is captured by ROOT which in turn updates histS. the history itself is maintained as a integer valued signal, the integer indicating which substate of the Or-state is the most recent one. However, if we use a shared variable for keeping track of the history, there will be no need to sustain the integer valued signal used for that purpose. Below, we show the code part of ROOT which updates the history values.

```
module ROOT :
...
var x in
[% the below block exists for each Or-state with history
       every immediate newhistS
              x := ?newhistS ;
              sustain histS(x) ;
       end
...
]
end module
```

3.6 Summary

In this chapter, we have discussed in detail the structure of the Statechart to ESTEREL code generator. We have discussed the structure of ESTEREL code generated by the translator. We have presented the general structure of the translation algorithms. We have also shown how the translation handles multilevel (T_4 type) transitions which is unique as such transitions are not supported in other variants of Statecharts.

Chapter 4

Visual Formalism for Communicating Reactive Systems

4.1 Introduction

A communicating reactive system is a class of reactive system that is composed of a collection of autonomous reactive nodes which communicate over *communication channels*. These nodes could be on different physical machines which exchange data through communication links. Altogether these nodes collectively and cooperatively achieve the overall functionality of the system. Many of the modern control systems fall into this class.

The design of reactive systems is known to be complex as compared to transformational systems. Statecharts[56, 57] which are an extension of finite state machines are used in the industry for modeling the behaviour of a reactive system. Visual formalisms like Statecharts are appealing to practicing software designers. Arguing the formal correctness however, is quite complex particularly when the number of states are large and hence, they need automated verification support.

The Statecharts in the form described in previous chapters do not support modeling of communicating reactive systems. The communication mechanism in Statecharts is based on shared variables and does not support communication through channels. In this chapter, we describe an extension [66] of Statecharts by introducing new states for handling communication through buffered channel or unbuffered channels. The extended notation is called *Communicating Statecharts (CS)* which is inspired by *Communicating Reactive Processes* [12]. The primitives can model synchronous as well as asynchronous communication. Each node in CS is modeled as a Statechart with additional communicating states showing the communication through channels. The operational semantics of CS as defined here preserves the *Step* semantics of Statecharts[57].

In the previous chapters, we had described translation [18, 103, 13]of Statecharts into ESTEREL [12] for verification and subsequent code generation. In this chapter, we describe an alternate approach of translating CS into Promela, the input modeling language of Spin model Checker [60] for verification. This is because ESTEREL does not have constructs to model communication through channels, which is required at the specification level. Using the CS to Promela translation tool (CSPROM), one can translate a CS specification into Promela and later use `Spin` model checking tool to verify the temporal properties of the system. We also show how one could model distributive algorithms in the tool. Subsequently we show [19] how to translate the communication primitives into ESTEREL so that it can be integrated into the ESTEREL based tool chain.

As an illustrative example, we have modeled the well known *Leader Election Protocol* used in distributed systems using CS notation. The model was translated into Promela using the CSPROM tool and we have used the translated model in Promela to show the correctness of the algorithm by verifying the known properties of the algorithm. The verification was carried out using the `Spin` model checker.

The contribution of this chapter is in extending the powerful visual formalism of Statecharts with features required to model distributed systems, define a scheme for formal verification of the model using a model checker, providing schemes to integrate ESTEREL tools for code generation and packaging them in a tool environment.

The chapter is organized as follows: The syntax and semantics of the communication primitives included in Statecharts are described in section 4.2 followed by the he model transformation scheme for verification in section 4.3. The code generation scheme for ESTEREL is discussed in section 4.5.

4.2 Communicating Statecharts

A distributed reactive system is modeled in CS as a network of independent reactive programs or nodes, N_i, each node having its own reactive interface with separate input/output signals and its own notion of instants. Each node N_i in CS is modeled as a Statechart with new communicating states showing the communication through channels. The nodes can communicate with each other using rendezvous states or buffered channels. The mechanism of rendezvous states uses unbuffered channel and the communicating state is called *Synchronous Communicating State*. On the other hand communicating state using buffered channel is called *Asynchronous Communicating State*. They are described in detail below.

4.2.1 Synchronous Communication State

A synchronous communication state or a rendezvous state indicates communication with the matched rendezvous state in other node via an unbuffered channel. The channel is specified as the label in rendezvous state. Figure 4.1 shows the representation of these states in a 2 node system in CS.

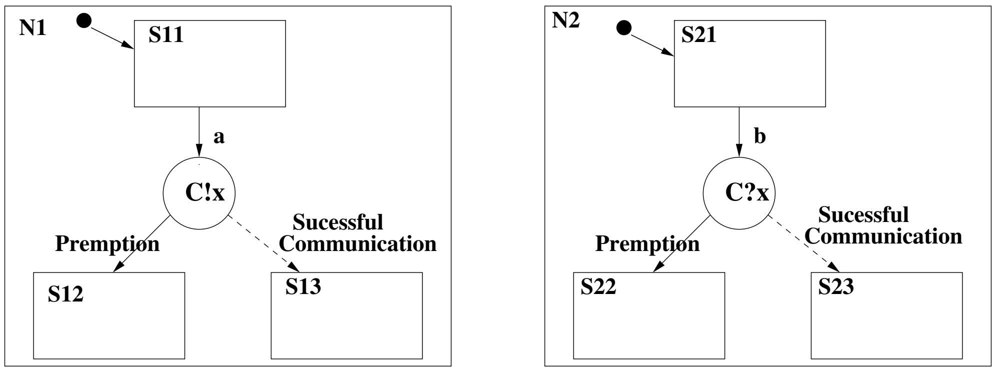


Figure 4.1: Rendezvous sender and receiver states

The syntax for labeling rendezvous state is `C ! | ? <mtype#> x` where "C" is the name of the channel through which the communication is done,"x" is the message being sent or the variable in which message is being sent, "?" implies that this is the receiver end of the channel C, "!" implies that this is the sender end of the channel C. The optional *mtype#* is the message type for the message "x". The sender does not send a message unless the receiver is ready to receive the message. There can be two types of exits from a rendezvous state. The transition labeled *Successful communication* is the exit from rendezvous state when communication succeeds and it is represented by a dashed edge. The transition labeled *Preemption* is the preemptive exit from rendezvous state when the reactive computation results in a state transition, before the actual communication completes. The rendezvous state is a special state and cannot be refined like a normal state in Statecharts.

In the Fig.4.1, `N1` and `N2` are two nodes of a distributed system, each represented by a Statechart. The actual communication takes place between the nodes when both are at the state labeled as `C!x` and `C?x`. The intuitive executional semantics of a rendezvous state is as follows:

- If the sender enters the rendezvous state, it waits until receiver enters the matching rendezvous state and vice versa. Both the sender and receiver exit this state after successful communication.
- While waiting for synchronization preemptive exit can take place on sender or receiver which can be due to a time_out event.

In contrast to a normal Statechart state, a rendezvous state does not have history and static reactions.

4.2.2 Asynchronous Communication State

Here the communication is via a buffered channel specified as the label of asynchronous communication state. Every channel has a message queue associated with it and messages are

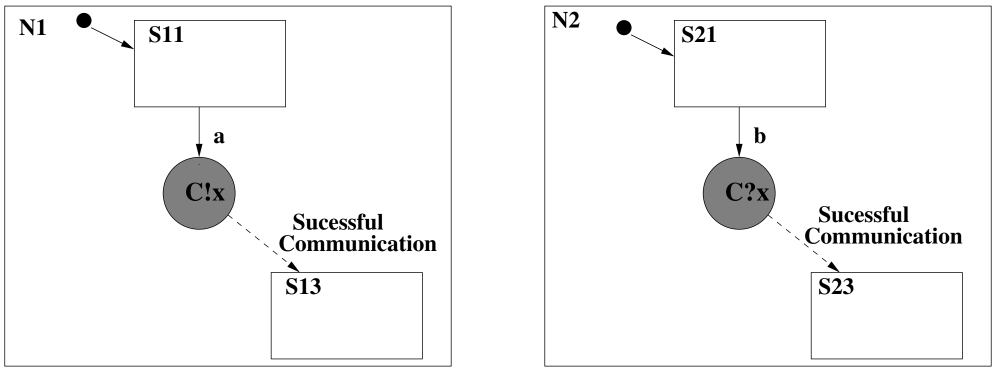


Figure 4.2: Asynchronous communication sender and receiver states

stored and retrieved in FIFO fashion. The sender doesn't have to wait for receiver to receive but puts the message in the message queue and continues with its computation. The sender is blocked only when the message queue is full. On the other hand the receiver picks up the message from the message queue and continues with its computation. The receiver gets blocked only when the message queue is empty. Asynchronous communication state is drawn as a shaded circle as shown in figure 4.2. Here the two nodes while in states S12 and S21 check at each instant for the event a and b in conjunction with the implicit guards as shown in the fig.4.2. Asynchronous communicating state has only one type of exit i.e., after performing successful communication.

Its intuitive executional semantics is explained below.

- Sender checks whether message queue is full before entering the communication state. If it is not full, it puts the message at the end of message queue and takes the non-preemptive transition in the next reaction. If the queue is full sender waits until the message can be put on the queue. The wait is always outside this state, essentially in the source state of incoming transition.
- Receiver checks whether message queue is empty before entering the communication state. If it is not empty, it reads the message from the message queue in FIFO fashion and takes the non-preemptive transition in the next reaction. If the message queue is empty, receiver waits until the message can be read from the queue. The wait is always outside this state, essentially in the source state of incoming transition.
- The waits on the message queue is defined as implicit guards on the incoming transition to this state. Thus for the asynchronous communicating states shown in figure 4.2 the guard on the sender side is *Not(C_ full)* and on the receiver side the guard is *Not(C_ empty)*.
- There is an upper bound on message queue length.

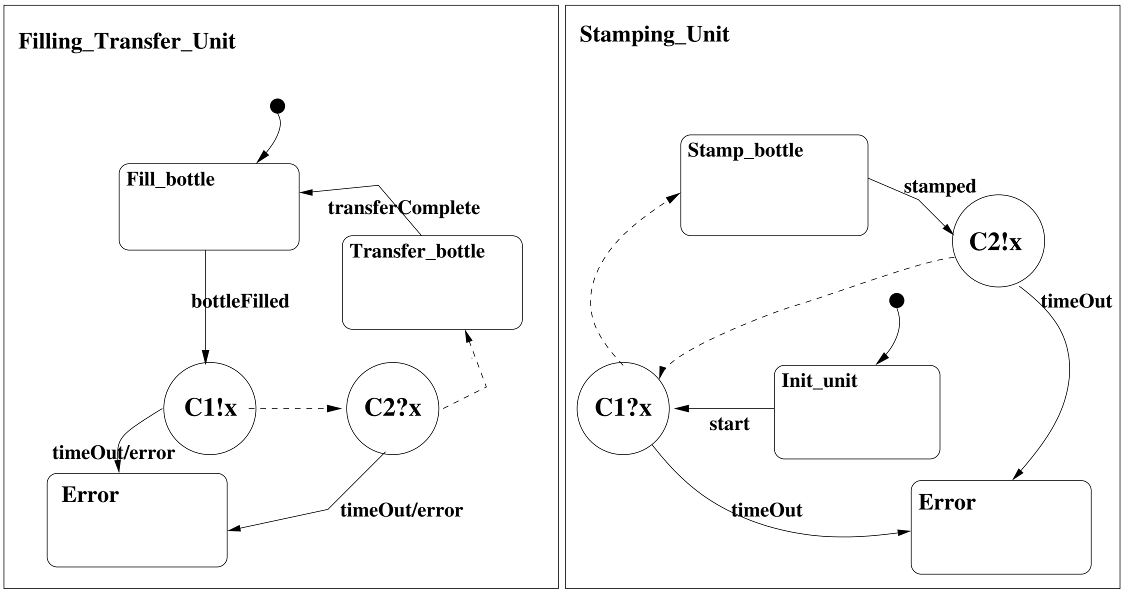


Figure 4.3: A Simple Bottling Plant

The asynchronous communication state does not have history and static reactions and cannot be refined further.

4.2.3 Modeling in CS: An Example of a Simple Plant Controller

The model in figure 4.3 depicts a simple bottling plant with two independent units namely filling bottles and stamping the bottles. Each unit is controlled by a separate controller. The filling unit controller needs to communicate with the stamping unit controller for synchronizing the stamping operation of a filled bottle. We have used synchronous communicating states to model the *lock step* operation of the system.

Both the nodes have been drawn with a very high degree of abstraction. The example shows the outer structure and how controller can communicate. In the *Filling_ Transfer_ Unit* the starting state is *Fill_ bottle*. After the bottle is filled indicated by the event *bottleFilled* the present state moves to communicating state *C1!x* which initiates communication with the *Stamping_ Unit* and waits for synchronization. When *Filling_ Unit* is in state *C1!x*, it can be preempted by a *timeOut* event *error* which moves the unit to an *Error* state for operator intervention. When the communication is successful, it waits for synchronization from the *Stamping _unit* at the *C2?x* and after successful stamping the *moveBottle* event is generated and the *Fill_ bottle* node moves to *Transfer_ bottle* state. From this state it makes transition to initial state on *transferComplete* event.

In *Stamping_ Unit* the initial state is *Init_ unit*. When it receives event *start*, it moves to communicating state *C1?x* and waits for communication indicating that the bottle is filled and ready to be stamped. If the synchronization is successful, it moves to the state *Stamp_bottle* where actual stamping of the bottle takes place and communicates the fact in the state *C2!x*. On successful completion of communication it goes back to *C1?x* state and waits for the next bottle. In case of *timeOut*, while waiting for synchronization the unit moves to *Error* state.

4.3 Model Verification for Safety & Liveness

Each node of CS is mapped as a process `proctype` in Promela. Thus if there are "n" nodes in a CS model there will be "n" instances of Promela `proctypes`. Each process in Promela representing the node in CS is logically composed of two modules namely *Environment* (this is also the main process) and *Reactive Kernel*. The *Environment* invokes the *Reactive Kernel*. The job of the *Environment* is to set the input signals and call the *Reactive Kernel* module to react on these signals as shown below.

```
Environment()
begin
    every step do
        set input signals
        set internal signals and outputs generated in the last step
        call reactive kernel
    end
end
```

Here the `step` is part of the operational semantics of Statecharts [57] and is associated with any event (external input events or internal events). The *reactive kernel* reacts to the signals by taking all possible transitions which are enabled from the set of active states. All state changes and output signals generated are returned to the *environment*. All reactions of the *reactive kernel* are atomic and take zero time (in reality take a bounded and a priori known amount of time, which is negligible compared to the frequency of reactions). This is the synchrony hypothesis and is the basis of all reactive models. *Reactive Kernel* waits for the *environment* to trigger the reaction. Depending upon the current state and the external input signals the *reactive kernel* reacts by taking all possible transitions. Internal signals may be emitted or set during the reaction and those add to the set of input signals in the next reaction. Output signals generated will be active only in the next reaction.

4.3.1 Scheme for Verification

We have implemented the scheme discussed above in a translator named CSPROM (Communicating Statecharts to Promela) that translates CS models to functionally equivalent Promela code. The translated code can be formally verified against the requirement using SPIN.

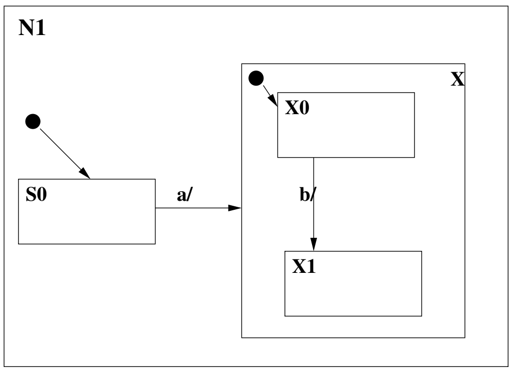


Figure 4.4: Simple Example showing model translation

As explained in section 4.3, every node of the CS is mapped to a process in Promela with two components, the main *Environment* process and the *reactive-kernel* as a procedure. Let us consider a single node of CS shown in Figure 4.4 and study its translation. Here S0 is the initial state of the system and X is a hierarchical state with two states X1 and X2. Signal a and b are two input signals. The *environment* code for this is shown below

```
/* This is the Promela code for environment
 S0_N1, X_N1, X1_N1, X2_N2 are the
 state variables, representing the states
 a_N1, b_N1 represent the boolean events
 in the context of the system N1
 */
active proctype N1_Env()
{
    S0_N1 = true;
    do
    ::  if
        :: a_N1 = 0
        :: a_N1 = 1
        fi;
        if
        :: b_N1 = 0
        :: b_N1 = 1
        fi;
        atomic
        {
            N1_kernel();
        }
    od
}
```

The *environment* first sets the initial state and then non-deterministically chooses the status of input signals and after that it makes an atomic call to *reactive_kernel* called N1_kernel.

The *kernel* first initializes the temporary state variables and initiates a do loop. Every state that has an outgoing transition has a block of code associated with it. In this example states S0 and X1 have the corresponding block of codes. If state S0 or X1 is enabled, the transition_condition is checked and if that is true state variables are set for the new configuration. The loop terminates when all the enabled transitions are taken and the next set of active states are assigned in hierarchical order.

```
/*  This is the Promela code modeling
the kernel/
inline N1_kernel()
{
    t_X_N1 = X_N1;
    t_S0_N1 = S0_N1;
    t_X1_N1 = X1_N1;
    t_X2_N1 = X2_N1;
    reaction_not_completed = true;
    do
    ::  (reaction_not_completed == true) →
        reaction_not_completed = false;
        if
        ::  (S0_N1 == true) →
            if
            ::  (a_N1 == true) →
                t_S0_N1 = false;
                S0_N1 = false;
                t_X_N1 = true;
                t_X1_N1 = true;
                reaction_not_completed = true;
            ::  else → skip
            fi
        ::  else → skip
        fi;
        if
        ::  (X1_N1 == true) →
            if
            ::  (b_N1 == true) →
                t_X1_N1 = false;
                X1_N1 = false;
                t_X2_N1 = true;
                reaction_not_completed = true;
            ::  else → skip
            fi
            :: else → skip
            fi
        :: else → break
        od;
        S0_N1 = t_S0_N1;
        X_N1 = t_X_N1;
        if
        :: (X_N1 == true) →
            X1_N1 = t_X1_N1;
            X2_N1 = t_X2_N1
        :: else →
            X1_N1 = false;
            X2_N1 = false
        fi
    }
```

4.4 Model of 3-node leader election ring

As an illustrative example we consider the well known algorithm for leader election in a unidirectional ring [60]. In this algorithm, each node sends a message with its *id*, to its right neighbour, and then waits for the message from its left neighbour. When it receives such a message, it checks the *id*, in this message. If the *id*, is greater than its own *id*, it forwards the message to the right; otherwise it swallows the message and does not forward it. If a node receives a message with its own *id*, it declares itself the leader by sending a termination message to its right neighbour and exits the algorithm as the leader. A node that receives a termination message forwards it to the right, and exits as non-leader.

Assuming there are three nodes in the ring. Each node has two channels, input and output, messages are read from input channel and written to output channel. The input channel of a node is shared as the output channel of left neighbour and output channel is shared as an input channel of right neighbour. In this algorithm, all processes will participate in the election. We consider a 3 node ring and fig. 4.5 gives the abstract view of the connectivity of the three nodes in a ring. A typical model of a node in CS is shown in fig. 4.6. The nodes are identified by the `node_id` value.

Here the state *Leader* indicates that when that particular node is in that state, it has become the leader. Similarly the state *Lost* indicates that the particular node has lost the election.

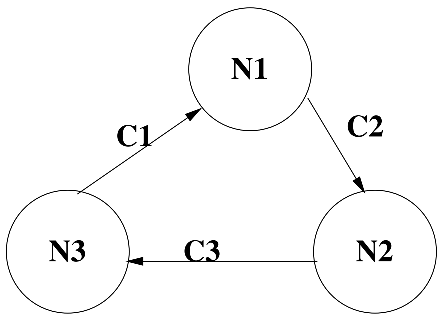


Figure 4.5: Leader Election Protocol: Nodes and channels connectivity

4.4.1 Verification for Correctness

We illustrate the verification process below by verifying two properties related to *Safety* and *Liveness* of the algorithm.

1. *Safety Property*

   The primary goal of the algorithm is that it must guarantee that at any point of time there is only one leader. The predicate $\forall i \in \{1, 2, 3\}.L\_N_i$ is true for the node which is the leader. Hence the proposition

   onlyonewinner $= ((\neg L\_N1 \wedge \neg L\_N2 \wedge L\_N3) \vee$
   $(\neg L\_N1 \wedge L\_N2 \wedge \neg L\_N3) \vee (L\_N1 \wedge \neg L\_N2 \wedge \neg L\_N3) \vee$
   $(\neg L\_N1 \wedge \neg L\_N2 \wedge \neg L\_N3))$ is true when there is only one leader. The variable L_N_i is set to true when the node $i \in \{1, 2, 3\}$ reaches the *Leader* state. The fourth term in this formula takes care of the states when there is no leader i.e during intermediate states of execution of the algorithm. As a safety property this should be true over all states i.e $[]onlyonewinner$ must be satisfied by the model. By running Spin model checker, it is found that the model satisfies this property.

   In case the model does not satisfy the property, Spin produces a counter example which can be traced in the Promela model by the guided simulator of the Spin user interface. The output also contains information about the state space of the model.

2. *Liveness Property*

   The algorithm starts in a state where there is no leader and as the algorithm terminates eventually a leader is selected. This property is stated in terms of internal state variables as

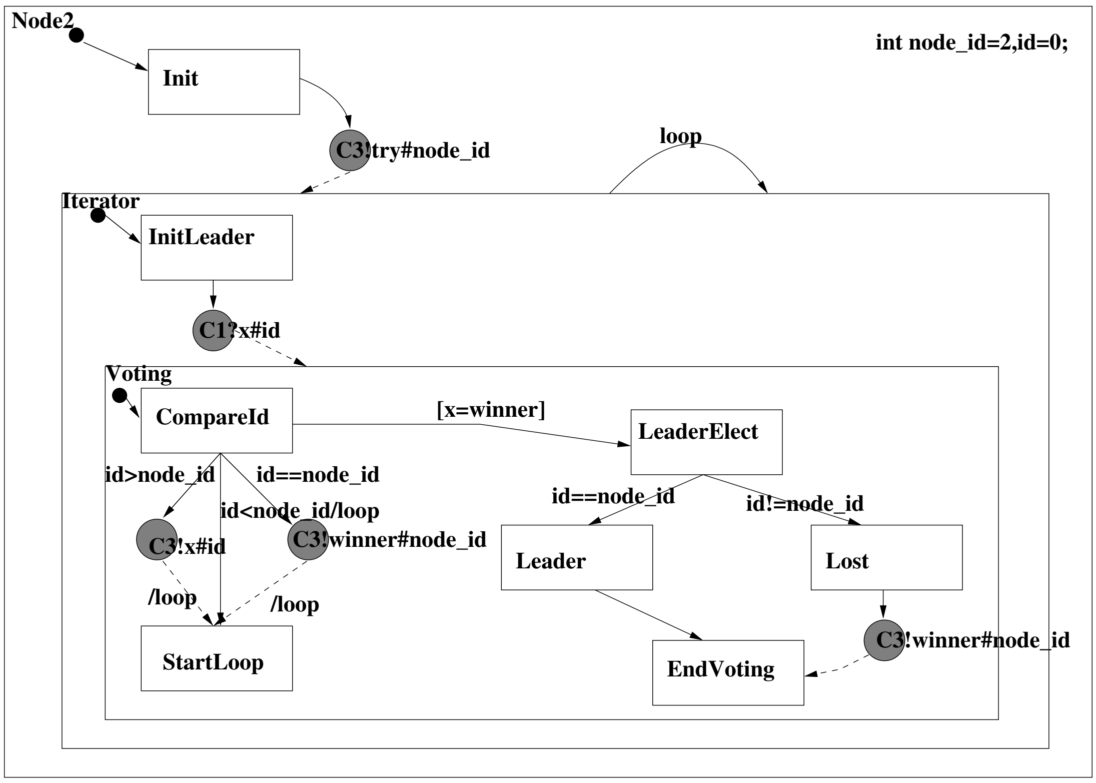


Figure 4.6: A typical node modeled in CS

$$[](noofwinner = 0) \longrightarrow <> (noofwinner = 1))$$

This property is also satisfied by the model. The variable `noofwinner` is a global state variable keeping a count of how many states have become the leader.

4.5 Model Code Generation

In previous chapters, we had presented a detailed algorithm for translation of Statecharts to Esterel constructs. It was shown how the different types of states and their composition could be compiled into Esterel . As explained earlier, the crux of the translation lies in

1. Extracting the hierarchy of states and transitions
2. Resolving the conflict in the transitions as per the Statemate semantics
3. Generating the code corresponding to the transitions between states
4. Generating code that models system state between transitions and

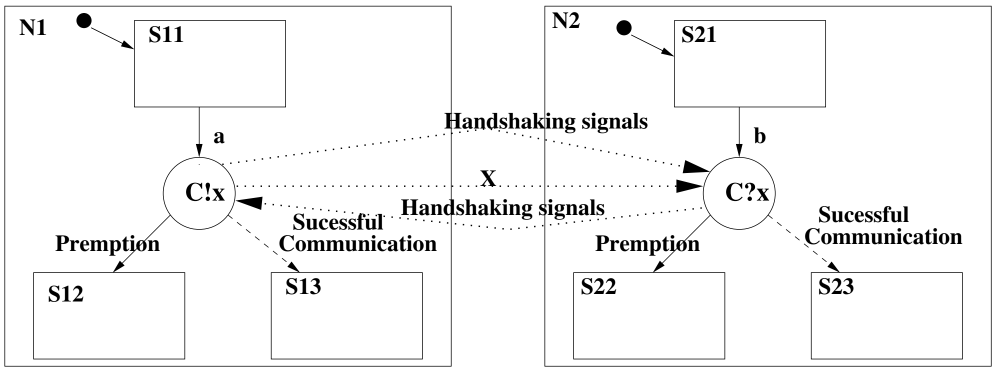


Figure 4.7: Rendezvous sender and receiver states

5. Generating code that supports communication via events and actions.

The advantage of converting the visual language into ESTEREL is in using tool chains available in synchronous languages. In this section we explain the scheme to be adopted in modeling the communication primitives shown in CS notations in ESTEREL . We only show the primitives for synchronous communication. We have implemented the rendezvous by using pure signals `labrt, rabrt, sabrt` etc., for local abortion and handshaking events indicating whether they are ready [6]. The ESTEREL code for handling states with communication shown in Fig.4.1 are explained below. The same figure annotated with the signals are shown in Fig. 4.7. The schematic code for the state N1 is given here.

```
module N1
   input a, rcpt,rabrt,labrt ;
   output x,conf,sabrt,success,fail ;
   % –signals—
   trap TN1 ∈ [
                 loop
                     [
                        await immediate goS11 ;
                        trap TS11 ∈ [
                                      %– code for state S1
                                      [   [ S11 ] ]
                                        ||
                                      await immediate a ;
                                      present labrt else
                                          emit x ;
                                          await case
                                              labrt :  emit sabrt ;
                                              emit fail
                                              rcpt : emit conf ;
```

```
                                            emit success
                                            rabrt :  emit fail
                                        end
                                    end present;
                                    await success ;
                                    await STEP ;
                                    emit sigS11_to_N1 ;
                                    exit TS11
                              ] end trap
                          present sigS11_to_ A then
                              emit go_S13
                            else present fail then
                                     emit go_S12%– preempted
                              end present
                          end present
                       ]
                  end loop
           end trap
       end
```

The sender node N1 works by first awaiting on the initiating event a and then emanating the signal `x` and then waiting for either an `rabrt` or `rcpt` signal to arrive from the receiving node. On receipt of the `rcpt` signal, the sender confirms the rendezvous via the `conf` signal. If on the other hand the sender is locally preempted indicated by the `labrt` signal, it responds by emitting the `sabrt` signal and terminates. Receipt of signal `rabrt` also causes unsuccessful termination of the rendezvous. A successful rendezvous is characterized by the presence of signal `success` at termination.

```
module N2
   input b, x,sabrt,conf,labrt ;
   output rcpt,rabrt,success,failure ;
   trap TN2 ∈ [
                 loop
                     [
                        await immediate goS21 ;
                        trap TS21 ∈ [
                                      %– code for state S21
                                      [   [ S21 ] ]
                                        ||
                                      await immediate b ;
                                      present labrt else
                                          await case
                                              labrt :  present x then
```

```
                                                      emit rabrt
                                                      emit failure
                                              end
                                              data : emit rcpt ;
                                              await case
                                                  labrt : emit failure
                                                  conf : emit success
                                                  sabrt : emit failure
                                              end
                                          end
                                      end present
                                      await STEP ;
                                      emit sigS21_to_N2 ;
                                      exit TS21
                              ] end trap
                          present sigS21_to_N2 then
                              emit go_S23
                            else present failure then
                                      %preempted
                                      emit go_S22
                              end
                          ]
                      end
                  ]
          end trap
      end
```

The receiver node works for by awaiting for the initiating event `b` and the waiting for a data `x` signal to arrive from sender node and then acknowledging it by emitting a `rcpt` signal. It then awaits the arrival of `conf` signal which will indicate successful termination.

4.6 Implementation of a Tool

We have developed a tool which has an environment to model, verify and subsequently generate ESTEREL code from visual model based on Statecharts. The tool has an editor (STATED), model verifier (CSPROM) and code generator generating code in ESTEREL (STATEST). The overall architecture of the tool is shown in Fig. 4.8. The editor is shown in Fig. 4.9. The CS model is created from the informal system/software description. Once the CS model is built, it can be translated into the Promela model using our CSPROM translator. This becomes the input to the Spin model checker. The system requirements are captured and formalized as formal specification during the model building process illustrated in Fig. 4.10. These formal specifications are put either as state assertions in the obtained Promela code or as temporal properties directly given in the Spin model checker.

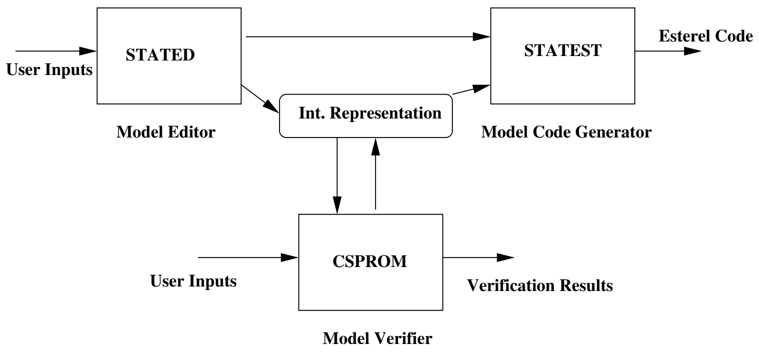
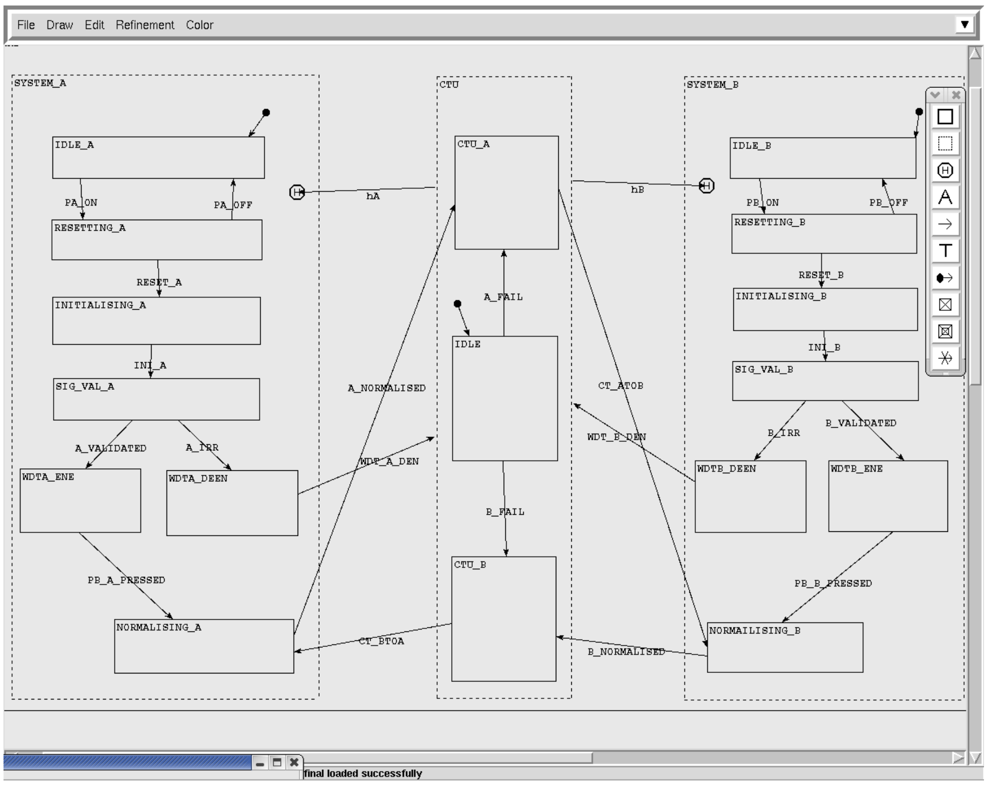


Figure 4.8: Tool Architecture



final loaded successfully

Figure 4.9: Tool Editor

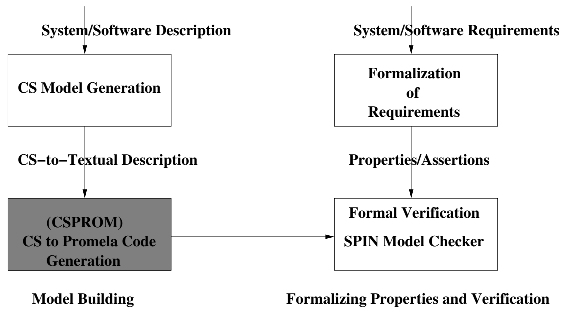

System/Software Description
CS Model Generation
CS–to–Textual Description
(CSPROM)
CS to Promela Code Generation
Model Building
System/Software Requirements
Formalization of Requirements
Properties/Assertions
Formal Verification
SPIN Model Checker
Formalizing Properties and Verification

Figure 4.10: The use of CSPROM in formal verification

The ESTEREL code generator module STATEST shares the intermediate representation with CSPROM and generates the ESTEREL code. It is possible to use verification tools from ESTEREL distribution like *Xeve* [21] and *EsterelStudio* [39].

4.7 Summary

The environment is planned to be extended with facility to incorporate host language code like C functions in the diagram to enable handling data dependent part of the actual design. We plan to extend the tool to generate directly C code from the model to be compiled and run on an operating system. The following points are worth mentioning regarding modeling and verification

- It is required to know some of the internal state variables to encode the temporal properties in LTL. We need to provide a framework in which it should be possible to express these properties at the level of abstraction of CS.
- Presently any counterexample generated by Spin during model checking for a property on the translated code has to be manually traced back to the CS model. It would be better, if the property it self can be specified as an observer in the CS notation and any counterexample generated by Spin model checker could be simulated in the CS model.

Chapter 5

Modeling Business Processes through Activity Diagram

5.1 Introduction

One of the important modeling artifacts used in UML, is the Activity Diagrams (referred as UML AD)that are used to model sequence of actions as part of the process flow. It is used to model sequence of actions to capture the process flow actions and its results. It focuses on the work performed in the implementation of an operation (a method), and the activities in a use case instance or in an object. Process flow modeling language like Activity Diagram has drawn attention in the recent years in terms of modeling workflows [108] and verification [41, 40]. In this chapter, we present a formal interpretation of Activity Diagrams in a process algebraic framework [23]. A simple activity diagram describing the order processing and account is shown in Fig. 5.1.

Although the OMG document [92] provides an intuitive semantics of Activity Diagrams, it lacks a formal semantics required for analysis and automatic code generation. Hence, in the recent past there has been a lot of interest in giving a formal semantics to Activity Diagrams.

In [109] it has been shown that UML AD can be used to model some of the workflows patterns identified in [108]. It is pertinent to ask *"Can we use UML AD to specify business processes which are prone to failure?"*. We opine that the traditional Activity Diagram needs to be enriched with additional constructs to enable us to model failures in any of the component processes. One of the advantages of having such a semantics for activity diagrams will allow modeling distributed workflows coupled with interruptible regions and evaluate their transitional state and behaviour for checking conformance to the requirement. Business Process Modeling Notation (BPMN) [93] has constructs to show failures and compensations, however we are not aware of a formal semantics of BPMN. On the other hand, the semantics of Activity Diagrams has been studied extensively in literature [42] and by enriching the constructs, it is expected to be useful.

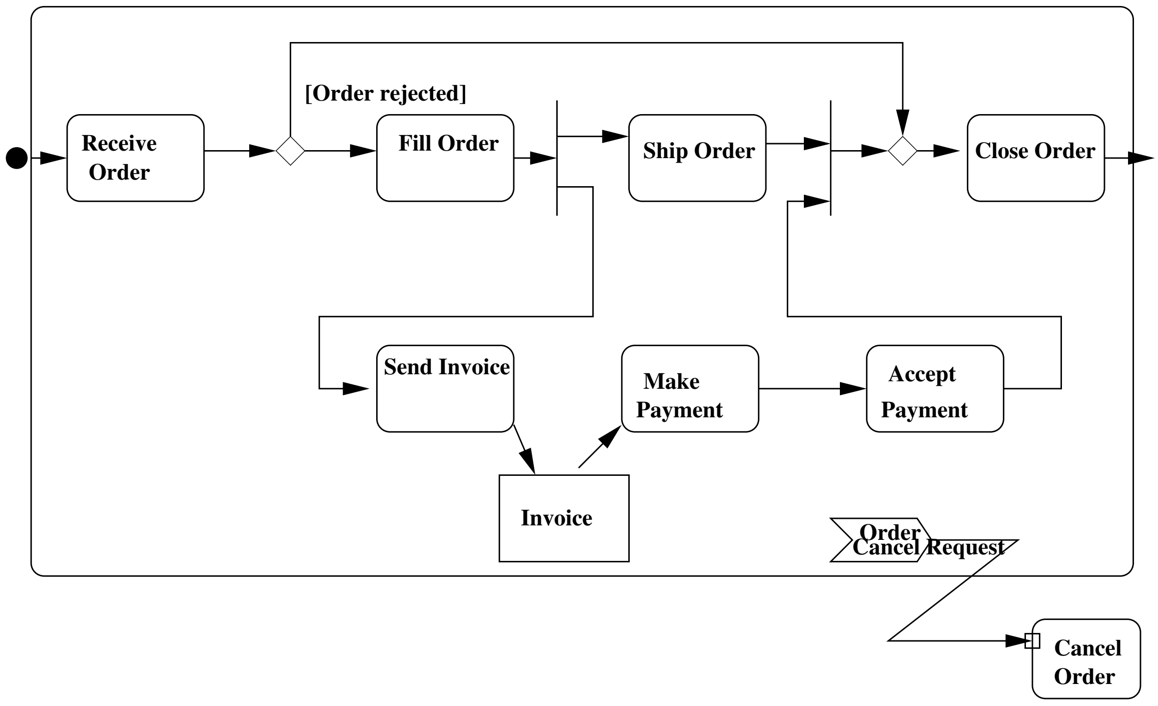


Figure 5.1: Simple Activity Diagram

The main contribution of this chapter is in the following

- Establishing the requirement of compensation actions in UML AD for modeling business processes.
- Enriching the constructs of UML AD with compensable actions [26] which enables modeling of failure in business processes.

The semantics of the additional constructs are based on CSP enrichment that can cater to failures in activities. Failures/exceptions are modeled as a part of the activities and is robust in the sense of CSP; as the failure action has also become an explicit action treated in a first-class manner and hence there is nothing like a real "exception". The proposed constructs can also be used to provide a theoretical framework for BPMN.

The chapter is organized with a description of various constructs of activity diagrams and their interpretation in a process algebraic notation are given in section 5.2. A brief insight into the requirement of activity diagram to model business logic is given in 5.3. Section 5.4 introduces the additional structures required to build compensating activities in modeling with a formal semantics.

5.2 Activity Diagrams: Interpretation in Process Algebra

An action is the fundamental unit of executable functionality in an activity [92]. The execution of an action represents some transformation or processing in the modeled system, which could be a computer system or a process. An action may have sets of incoming and outgoing activity edges that specify control flow and data flow from and to other nodes. An action will not begin execution until all of its input conditions are satisfied. The completion of the execution of an action may enable the execution of a set of successor nodes and actions that take their inputs from the outputs of the action. The sequencing of actions are controlled by control edges and object flow edges within activities, which carry control and object events respectively. An action can only begin execution when all incoming events are present. An action execution represents the run-time behavior of executing an action within a specific activity execution. When the execution of an action is complete, it offers events in its outgoing control edges, where they are accessible to other actions. Communicating Sequential Processes (CSP) [23] is a process algebra which is suited for modeling such process flow systems. One of the advantage of CSP is that one can make assertions about safety and correctness properties based on traces.

An interpretation of UML AD activity as CSP [23] processes is given below. The basic flow of activity is defined by the process PROC(P) as

$$\alpha PROC(P) = \{P.entry, P.in, P.exit\}$$
$$PROC(P) = P.entry \rightarrow P.in \rightarrow P.exit \rightarrow SKIP$$

The process PROC(P) first performs the event *P.entry* at the start representing the start of the activity P, the event *P.in* represents the activity P is being performed and *P.exit* represents the completion of activity. This is similar to the basic events considered in each state in a Statechart [57]. In addition to these events, we consider two more events: $\surd$ to notify the upper level activities about the completion of the activities in the present scope, $\dagger$ to notify a general trigger to higher level activities. An activity diagram is constructed as a legal combination of activity, start, stop state elements and merge, decision, fork and join relationship elements.

5.2.1 Synchronisations in Various Control Flow Patterns

A number of control flow patterns and their ability to be modeled in UML AD has been identified in [109], which define elementary aspects of control flow. These are also used as elementary control-flow in Workflow Management.

We provide the transition rules [23] of some of the basic control patterns shown in Fig.5.2 as processes in CSP. Σ denotes the set of observable events in the environment and the terminal events $\Omega = \{\surd, \dagger, ©\}$ represent the different ways in which an activity may terminate.

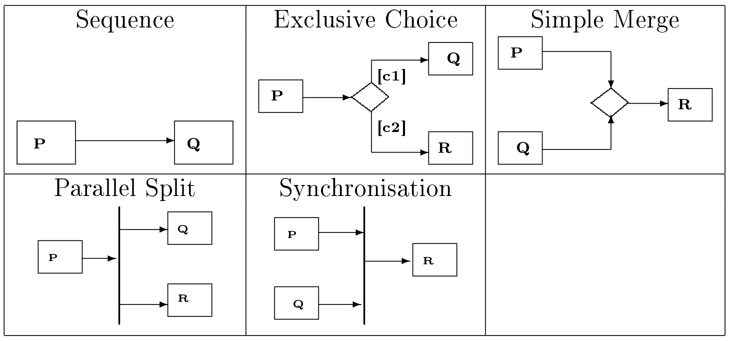


Figure 5.2: Basic Constructs of Activities

Successful termination is represented by the $\surd$ event, action denoted as $P\xrightarrow{\surd}\circ$. An interrupt event is represented by the $\dagger$ event and yielding is represented by the ©event. The interrupt and yielding events are described later.

Sequence We consider three cases
Process P does not terminate

$$\frac{P\xrightarrow{\alpha}P'}{P;\ Q \xrightarrow{\alpha} P';\ Q}\alpha \in \Sigma$$

Process P terminates normally

$$\frac{P\xrightarrow{\surd}0 \wedge Q\xrightarrow{\alpha} Q'}{P;\ Q \xrightarrow{\alpha} Q'}\alpha \in \Sigma \cup \Omega$$

Process P Terminates abnormally

$$\frac{P\xrightarrow{\omega}0}{P;\ Q\xrightarrow{\omega}0}\omega \in (\Omega - \{\surd\})$$

Exclusive Choice Condition $c_1 \subset \alpha$

$$\frac{P\xrightarrow{\surd}0 \wedge Q\xrightarrow{\alpha} Q'}{P;\ Q\Box R \xrightarrow{\alpha} Q'}\alpha \in \Sigma$$

Condition $c_2 \subset \alpha$

$$\frac{P\xrightarrow{\surd}0 \wedge R\xrightarrow{\alpha} R'}{P;\ Q\Box R \xrightarrow{\alpha} R'}\alpha \in \Sigma$$

Parallel Split

$$P;\ Q \parallel R$$

$$\frac{P\xrightarrow{\checkmark}0 \wedge Q\xrightarrow{\alpha} Q' \wedge R\xrightarrow{\alpha} R}{P;\ Q \parallel R \xrightarrow{\alpha} Q' \parallel R}\alpha \in \alpha Q$$

$$\frac{P\xrightarrow{\checkmark}0 \wedge Q\xrightarrow{\alpha} Q \wedge R\xrightarrow{\alpha} R'}{P;\ Q \parallel R \xrightarrow{\alpha} Q \parallel R'}\alpha \in \alpha R$$

The rules for *simple merge* and *synchronization* are expressed in terms of the above rules.
Simple Merge

$$P\Box Q;\ R \equiv (P;\ R)\Box(Q;\ R)$$

Synchronization

$$P \parallel Q;\ R$$

5.2.2 Handling Interrupts

Although not shown in [108], it is possible to break the flow in an activity diagram by an asynchronous interrupt event. If activities are sequential, the exception causes an immediate transfer of the flow of control. An interrupt handler may be used to catch interrupts: in $P \bigtriangleup Q$, an interrupt caused in P triggers execution of the handler Q. It follows that a trace of $(P \bigtriangleup Q)$ is just a trace of P up to an arbitrary point when the interrupt occurs, followed by any trace of Q.

$$\alpha(P \bigtriangleup Q) = \alpha P \cup \alpha Q$$
$$traces[\![P \bigtriangleup Q]\!] = \{s \frown \langle\dagger\rangle \frown t \mid s \in traces[\![P]\!] \wedge t \in traces[\![Q]\!]$$

The control flows to interrupt handler from the first process which is caused by the † event as shown below:

$$\frac{P\xrightarrow{\alpha}P'}{P \bigtriangleup Q \xrightarrow{\alpha} P' \bigtriangleup Q}\alpha \in \Sigma$$

$$\frac{P\xrightarrow{\dagger}0 \wedge Q\xrightarrow{\dagger} Q'}{P \bigtriangleup Q \xrightarrow{\dagger} Q'}\dagger \notin \Sigma$$

$$\frac{P\xrightarrow{\checkmark}0}{P \bigtriangleup Q\xrightarrow{\checkmark}0}$$

Fig. 5.3, shows how exceptions can be raised in activities and in CSP and it can be written as $ProcessOrder \bigtriangleup CancelOrder$. If interrupts could be nested i.e if $P \bigtriangleup Q$ could be interrupted and R is the handler then it could be specified as $(P \bigtriangleup Q) \bigtriangleup R)$. Generating an interrupt inside the activity diagram to interrupt other activities is shown by a special process called *THROW* which generates the interrupt event † and terminates.

$$\frac{}{THROW \xrightarrow{\dagger} 0}$$

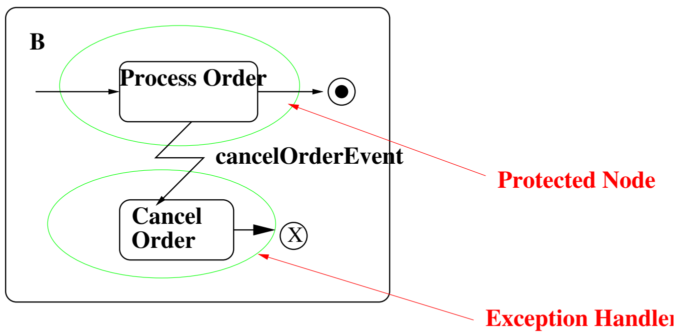


Figure 5.3: Activity with Exception Handlers

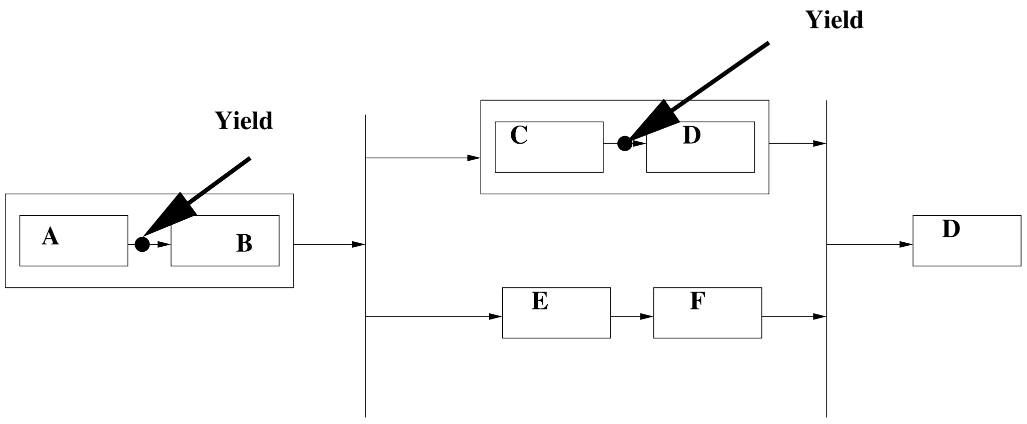


Figure 5.4: Activity with Yielding Points

On occurrence of the interrupt event control flow in the activity represented by process P will be transferred to Q.

An activity, which is ready to yield to an interrupt is to be indicated by local checkpoint where the state of the process is known and can be restarted. The behaviour of such activities is similar to check-pointed transaction. A local snapshot of the activity as shown in fig. 5.4 is taken through the primitive YIELD operator so that P YIELD Q means run P with checkpoint Q. This is similar to the Hoare's *restart with checkpoint* operator [85]. The transition rules are specified as

$$\frac{P \xrightarrow{\alpha} P'}{P \text{ YIELD } Q \xrightarrow{\alpha} P' \text{ YIELD } Q} \alpha \in \Sigma$$

$$\frac{}{P \text{ YIELD } Q \xrightarrow{\dagger} Q \text{ YIELD } Q}$$

$$\frac{}{P \text{ YIELD } Q \xrightarrow{\copyright} P \text{ YIELD } P}$$

5.3 Modeling Business Processes as Activity Diagram

Let us now consider the activity diagram shown in Fig. 5.5, where activity B can fail. This activity B can be a service provided by a server. Since the UML AD can only model forward flow, it is not possible to show the actions required if the service provided by this activity fails.

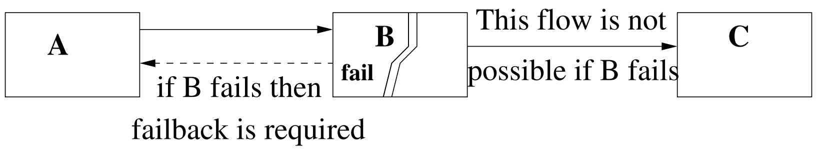


Figure 5.5: Activity with Failure

If the activity A is not successful because of some internal exception or an external condition, we must be able to undo the partial effect of actions executed in A. Let us now consider the activities required to process an order for which the activity diagram is shown in fig 5.6.

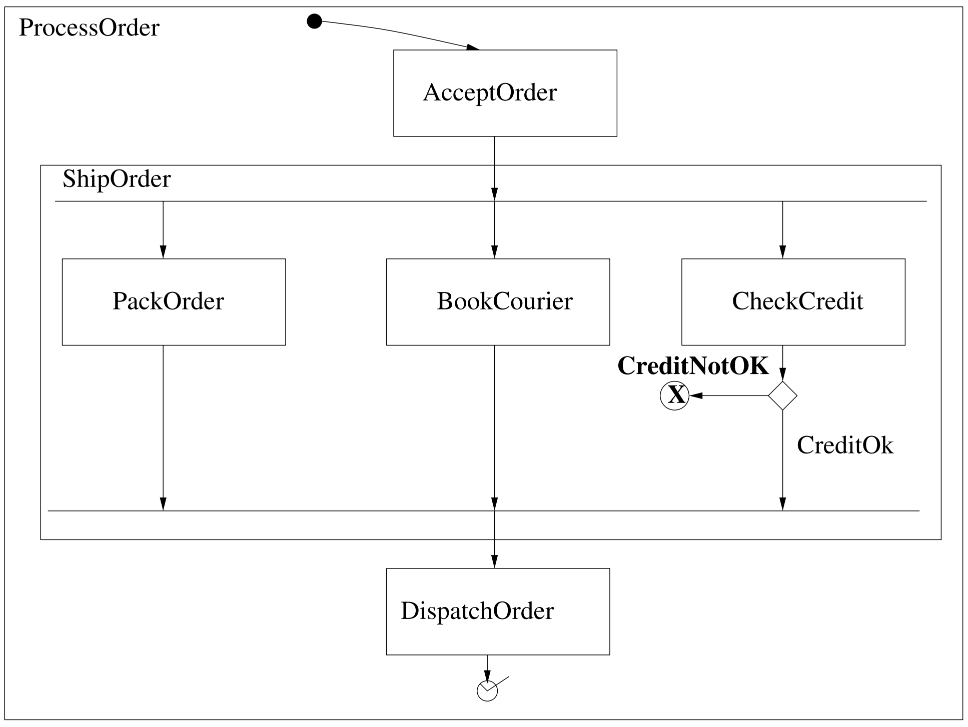


Figure 5.6: Order Processing

This is the classical book store problem where customers can order books over the web to the vendor. The vendor may not store all books and need some time in processing with other suppliers. However he can check the credit status of the customer with the bank. In case the activity `CheckCredit` reports a credit failure like `CreditNotOK` which should trigger the cancellation of the order, cannot be shown using the traditional token flow semantics of UML Activity diagrams. This is the underlying motivation.

A business process as described above typically consists of steps (each of which may be refined in substeps) and each step is called an activity. The requirements of business processes modeling are to be able to describe the process map showing the flow in the activities, description of these activities, handling exceptions and failure.

5.4 Modeling Failures in Activity Diagrams

We extend the syntax and semantics of UML activity diagram inspired by [107] and [26]. Here an activity is drawn as a box with two entry points and three exit points. The entries and exits are as shown in Fig.5.7. The box indicates an activity which may be composed of sub-activities but the interior components and connections of the box can be ignored from the outside. The entry and exit points of a compensable activity are activated in a standard sequential ordering. The normal entry point for an activity is at the start and failure leads to an exit along the exit labeled fail, which returns control to the compensation of the previous transaction. Successful execution ends with a finish, which will start the next activity in the sequence. If a subsequent activity fails, triggering a fail-back, this activity is able to compensate. If an activity detects that it can neither compensate nor succeed, it will allow the control to pass on the throw exit, which needs to be handled at the higher level as shown in Fig.5.8. After compensation, the activity exits by the failure arrow as before. In this sense, the compensable activities has a three way token flow as shown in Fig. 5.7.

For example, consider a simple activity whose input is X and which computes an output data Y such that $Y = X + X$. The compensating activity must be able to compute X such that $X = Y/2$. The action of compensation is to save a local snapshot of local state (values of variables) before change and restore it when required to compensate. This is the technique used in traditional transaction processing systems. The post condition of an activity at the finish edge must entail the precondition at the fail-back edge. $Post_{finish} \Rightarrow Pre_{failback}$. Similarly $Pre_{start} \Rightarrow Post_{fail}$.

This is similar to what is supported at procedural level in BPEL4WS where the compensation handler can be invoked by using the `compensate` activity.

```
<compensationHandler>
activity
</compensationHandler>

<compensate scope=''ncname'' ? attributes>
standard block
</compensate>
```

The advantage of graphical notation like Activity Diagram is that it would be easier to capture

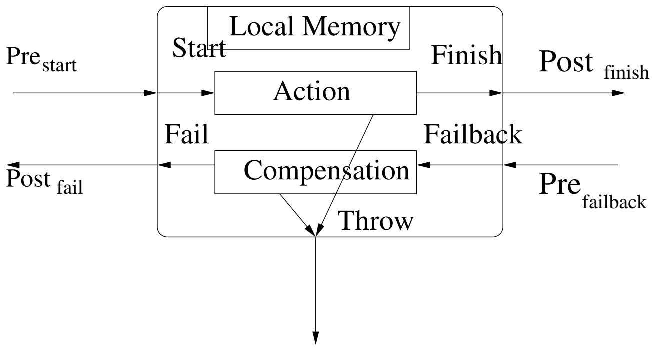


Figure 5.7: Activity with Compensation

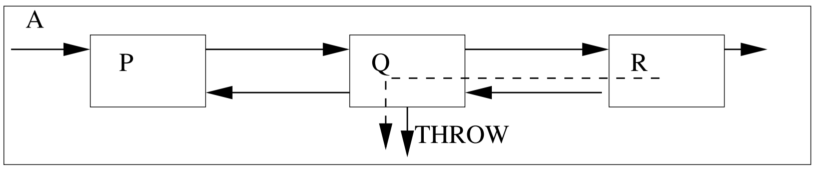


Figure 5.8: Composition of Compensating Activities

the choreography in a graphical formalism than in an imperative language like BPEL.

5.4.1 Semantics of Activity Diagrams with Failures in Enriched CSP Framework

In order to support failed activities, we use compensation operators [26] and the *Activities* are classified into *standard* and *compensable activities*. A compensable activity has associated compensation actions which are invoked in case of a failure in the forward activities. A compensable activity consists of a forward behaviour and a compensation behaviour. In the case of an exception, activities will be executed to compensate the forward behaviour. The basic way of constructing a compensable activity is through the compensation primitive $P \div \bar{P}$, where P is the forward activity and $\bar{P}$ is its associated compensation. $\bar{P}$ should be designed to compensate for the effect of P and may be run after P has completed.

The compensation enabled activity $PP = P \div \bar{P}$ is composed of two standard processes. The first one is called forward process which is executed during normal execution and the second one is called the compensation of the forward process which is stored for future use when it is required for compensation:

$$\frac{P \xrightarrow{\alpha} P'}{P \div \bar{P} \xrightarrow{\alpha} P' \div \bar{P}} \alpha \in \Sigma$$

If the forward activity terminates normally then the complete activity terminates with $\bar{P}$ a the result compensation. Hence at the end of successful termination of present activity, the compensating activity $\bar{P}$ is installed.

$$\frac{P \xrightarrow{\surd} 0}{P \div \bar{P} \xrightarrow{\surd} \bar{P}}$$

If any forward activity terminates abnormally, then so does the complete activity, resulting in an empty compensation activity

$$\frac{P \xrightarrow{\dagger} 0}{P \div \bar{P} \xrightarrow{\dagger} 0}$$

If the activity $PP = P \div \bar{P}$ cannot progress either way due to an internal condition, it generates a throw which should be caught by an exception handler. This handles the three way flow

$$PP \bigtriangleup I_P = [P \div \bar{P}] \xrightarrow{\dagger} I_P$$

$$traces[\![PP \bigtriangleup I_P]\!] = \{s \frown t \mid s \in traces[\![P \div \bar{P}]\!] \wedge t \in traces[\![I_P]\!]$$

A standard activity can be transformed into a compensable activity by adding to it an activity, which actually does nothing (SKIP). We use P,Q to identify standard activities and PP,QQ to identify compensable activities.

$$\boxed{\begin{array}{ll} \text{PP} & ::= P \div \bar{P} \text{(compensation pair)} \\ & \mid SKIPP = SKIP \div SKIP \\ & \mid THROWW = THROW \div SKIP \end{array}}$$

The parallel and sequential composition operators for compensable processes are designed in such a way that ensures that after the failure of an forward activity the necessary activities are performed in an appropriate order to compensate the effect of already performed actions. Sequential composition of compensable processes is defined so that the compensations for all performed actions will be in the reverse order to their original sequence. Let us consider $PP = P \div \bar{P}$ and $QQ = Q \div \bar{Q}$ as two compensable activities then the following rules define the sequential composition of compensating activities

$$\frac{PP \xrightarrow{\alpha} PP'}{PP;\ QQ \xrightarrow{\alpha} PP';\ QQ} \alpha \in \Sigma$$

if PP fails the whole activity terminates and the compensation activity of PP that is run.

$$\frac{PP \xrightarrow{\alpha} \bar{P}}{PP;\ QQ \xrightarrow{\alpha} \bar{P}} \alpha \in (\Omega - \{\surd\})$$

However if QQ terminates normally after PP, the compensation of PP i.e $\bar{P}$ should be composed with the compensations from QQ i.e $\bar{Q}$. The reversal of process order is shown by $\langle \bar{Q}, \bar{P} \rangle$. This is shown by

$$\frac{PP \xrightarrow{\surd} \bar{P} \wedge QQ \xrightarrow{\surd} \bar{Q}}{PP;\ QQ \xrightarrow{\omega} \bar{Q};\ \bar{P}} (\omega \in \Omega)$$

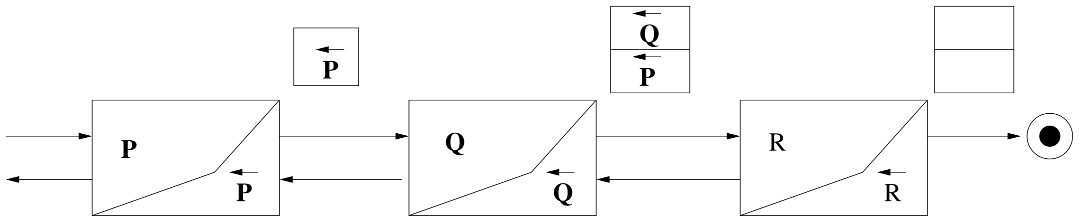


Figure 5.9: Stack of Compensation Activities

A compensable activity PP can be converted into standard activity by defining a block $[PP] = P \div \bar{P} \backslash \alpha P \cup \alpha \bar{P} \cup \dagger$. Successfully completed PP represents successful completion of the whole transaction block and compensations are no longer needed. When the forward behaviour of PP throws an interrupt, the compensations are executed in the appropriate order and the interrupt is not observable outside the block. Parallel composition of compensable activities is defined in such a way that compensations for performed actions will be accumulated in parallel. We assume that each of the activities P and Q are not raising interrupt and not yielding to interrupt.

$$[P \div \bar{P} \parallel Q \div \bar{Q};\ THROWW] = (P \parallel Q);\ (\bar{P} \parallel \bar{Q})$$
$$[P \div \bar{P} \parallel Q \div \bar{Q} \parallel THROWW] = SKIP \Box (P;\ \bar{P})$$
$$(Q;\ \bar{Q}) \Box (P \parallel Q);\ (\bar{P} \parallel \bar{Q})$$

A typical behaviour concerning the stack of compensation activities is shown in Fig. 5.9. One of the safety requirement of such compensating activity diagram is that the stack of compensating activities must be empty at the end. Now let us consider the above activity diagram in Fig.5.6 to process orders which require compensation because of exceptions raised by the `CheckCredit` activity when sufficient credit does not exist. The modified activity diagram is shown in Fig 5.10. The activities can be specified formally as

$$\begin{aligned} \text{ProcessOrder} &= (\text{AcceptOrder} \div \text{CancelOrder}) \\ &\quad ;\ \text{ShipOrder};\ \text{DispatchOrder} \\ \text{ShipOrder} &= (\text{PackOrder} \div \text{UnpackOrder}) \parallel \\ &\quad (\text{BookCourier} \div \text{CancelCourier}) \\ &\quad \parallel \text{CheckCredit} ;\ (\text{CreditOK} ;\ \text{SKIPP} \Box \\ &\quad \text{CreditNotOK} ;\text{THROWW}) \end{aligned}$$

This shows the underlying formal description of the activity diagram with compensating constructs. The advantage is in that this can be subjected to analysis for showing certain desired properties of business logic. The model can be used also to construct an implementation from the description like that of [95].

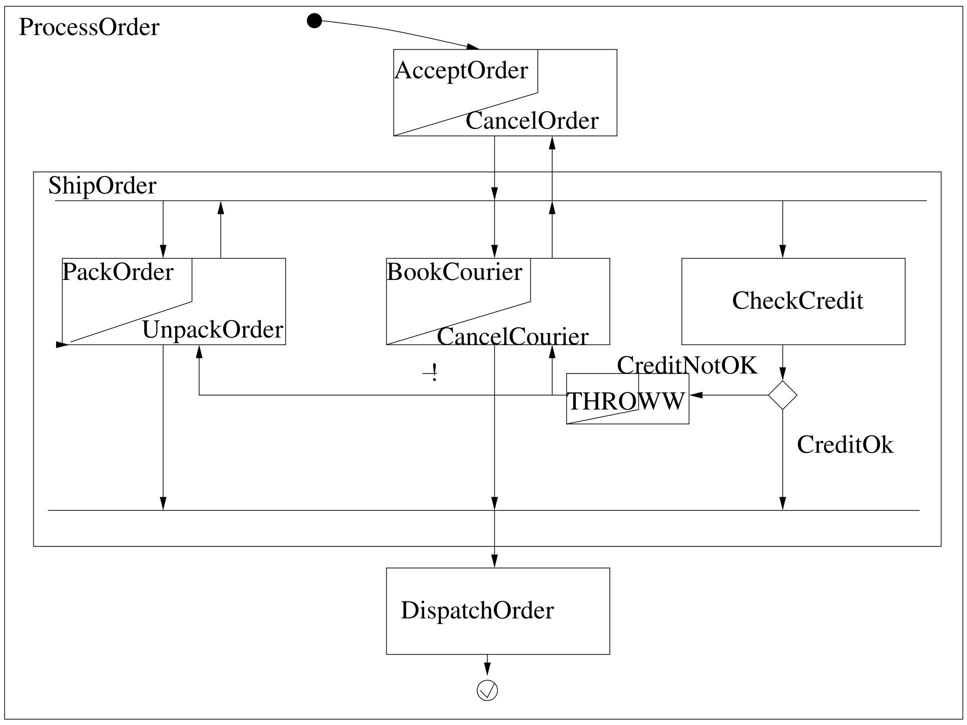


Figure 5.10: Activity with Compensation

5.5 Summary

In this chapter, we have presented the semantics of UML Activity Diagrams in a process algebraic framework using CSP as the formal language. We discussed the requirement of compensations in Activity Diagrams which can be used to model business process logic. Later we have presented the main contribution of this chapter: a semantics of compensable Activity Diagrams using the framework of flow compensable process languages.

Chapter 6

Implementation of Activity Diagrams in ESTEREL

6.1 Introduction

In this chapter, we extend the process algebraic semantics of activity diagrams and propose a reactive formalism [15] of Activity Diagrams of UML AD. The synchronous model for the Activity Diagrams is represented as a collection of transformation rules for each construct of the Activity Diagrams. We use ESTEREL [12] language for description purpose. Our approach combines the requirement level and implementation level semantics. Further the notion of procedure call transitions as used in activity diagrams are captured nicely through the ``run module'' construct and one can specify the number of incarnations of the same module when called multiple times. Since it is based on ESTEREL , that has efficient code generation tools, the transformations can be used to realize a system directly from the model. Thus in our approach, we can not only reason about functional requirements of UML AD but also generate validated code automatically. This approach is useful for model based design of embedded systems. In this presentation, we are concerned with the Intermediate Level of Activity Diagrams that include control and data flow and decisions.

The main contribution of this chapter is in establishing a semantic mapping between UML AD to a synchronous language which allows validated code generation. This chapter is organized as follows: The ESTEREL model of the basic activity constructs are provided in section 6.2. In section 6.5, a brief description of simulation and code generation based on the synchronous framework is presented. In section 6.7 a possible implementation of compensating activities is given in terms of Mode Automata. Verification approaches are presented in section 6.6.

6.2 Basic Constructs of Activity Diagram and their Implementation in Esterel

A basic `ActivityNode` is modeled by an Esterel module named after the node. The invocation of the activity is modeled by instantiating the module using the `run module` construct.

A basic ActivityNode can invoke an asynchronous task which can handle system specific functions and can be modeled by an Esterel task statement such as `exec taskA ()() return ExitA`, where taskA is the external process performing the actual action written in the host language. The completion of the task is signaled by emitting the signal `ExitA` referred as a return signal. A return signal cannot be internally emitted by the program. In our model we ignore the external action for the purpose of simplicity.

Each activity node has the following set of signals associated with it.

- `EntryS` is the signal emitted when a particular activity node is entered.
- `InS` is the signal emitted when an action in a particular activity node is being performed.
- `ExitS` is the signal emitted when a particular activity node is completed.

We also assume that there is a root activity node which contains and controls the sequencing of the activity nodes through the activity edges. In the example shown in Fig. 6.1 the module `simpleActivity` performs the task of passing control tokens from the activity `sendPayment` to the activity `receivePayment`. The activity node `simpleActivity` is the root activity controlling the activities `sendPayment` and `receivePayment`. The activities `sendPayment`, `receivePayment` and `simpleActivity` in the above example, can be interpreted through the Esterel fragments shown in the Fig.6.1.

6.2.1 Modeling Merge Node

A merge node (cf. Fig. 6.2) is a control node that brings together multiple alternate flows. It is not used to synchronize concurrent flows but to accept one among alternate flows. It has multiple incoming edges and a single outgoing edge. It can be described as follows

```
module mergeNode
   run  A% the module A implements activity A
    ||
   run  B% the module B implements activity B
    ||
   await  ExitA ;
```

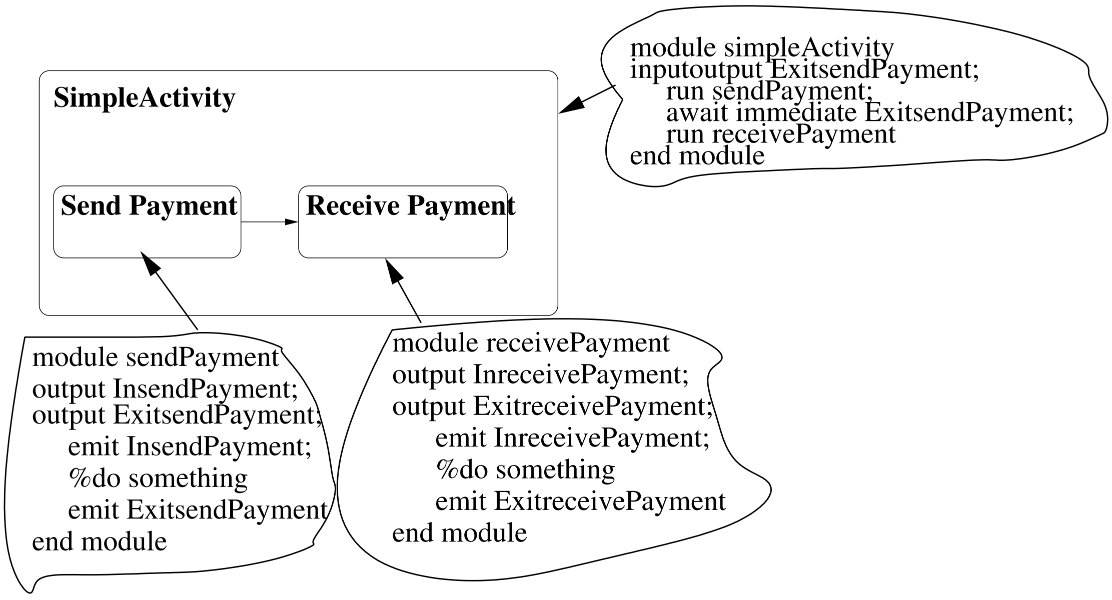


Figure 6.1: Simple node

```
    run  C% The module C implements activity C
     ||
    await  ExitB
    run  C% The module C implements activity C
end module
```

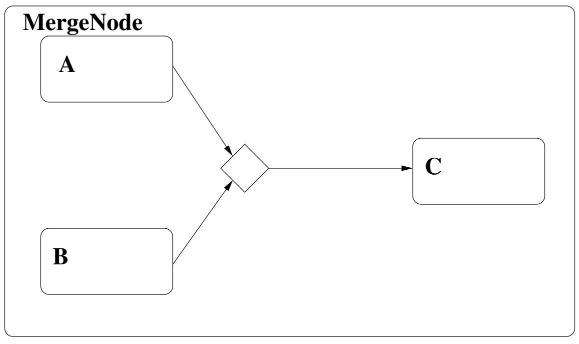


Figure 6.2: Merge Node

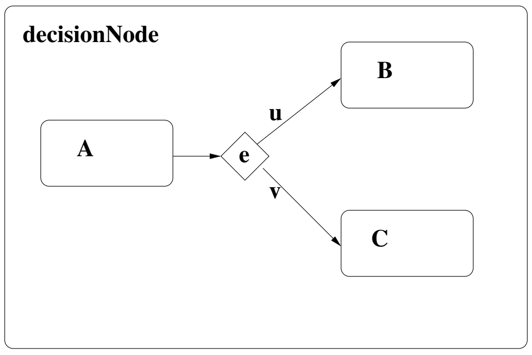


Figure 6.3: Decision Node

Here the activities A and B are started concurrently, but whichever activity completes earlier, starts the activity C. If activity A and B completes together, then two instances of C would be running at the same time. This interpretation is in line with recent OMG document [92].

6.2.2 Modeling Decision Node

A decision node (cf. Fig. 6.3) is a control node that chooses between the outgoing flows. It has one incoming edge and multiple outgoing edges. It can be described by the following ESTEREL fragment.

```
module decisionNode
    var e in
   run A ;
   if  e = u
      run B ; % e is the guard which if has value u then run B
     else if  e = v
             run C ; % e is the guard which if has value v then run C
       end
   end
end module
```

Here after the activity A completes, the control passes to activity B or C depending on the guard condition e being equal to u or v respectively.

6.2.3 Modeling ForkJoin Node

A forkJoin node (cf. Fig. 6.4) is a control node that splits a flow into multiple concurrent flows. It has one incoming edge and multiple outgoing edges. Tokens arriving at a fork node are duplicated across the outgoing edges. Tokens offered by the incoming edge are all offered to the outgoing edges.

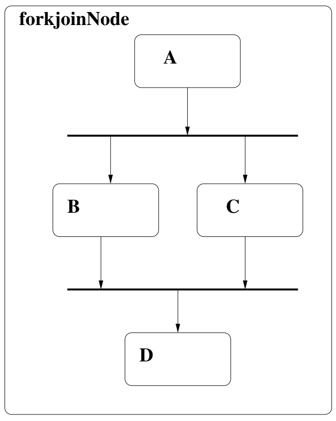


Figure 6.4: Fork Join Node

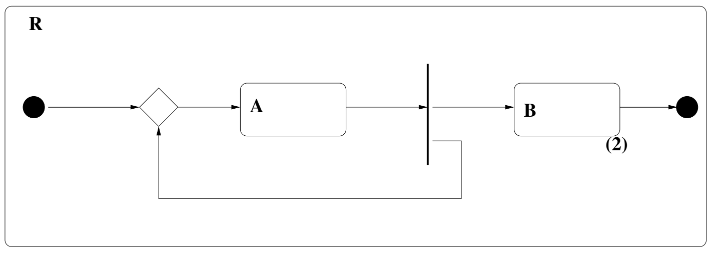


Figure 6.5: Reentrant Node

The forking and joining of activities can be described by the following ESTEREL fragment.

```
module forkJoinNode
    run  A% run activity A
       [
        run  B% run activity B
          ||
        run  C% run activity C
        ]
    run  D% run activity D
end module
```

Here after the activity A completes the activities B and C are started concurrently. Once both of B and C are complete, D is started. If concurrent activities are not modeled carefully this may lead to problem. Let us consider the case as shown in the Fig. 6.5. Here completion of A forks A once again with B. Thus, a possible run of the system is $A \rightarrow AB \rightarrow ABB \rightarrow \cdots$. That is there can be an infinite incarnation of B. This causes problem with verification because of unboundedness of states.

If we need to consider finite number of instances, we can use the parallel construct in ESTEREL to specify a finite number of concurrent activities. This is an advantage of the model, where one can specify the number of instances of the same activity which could be forked simultaneously. This closely maps to Workflow Management Systems, where one would specify the maximum number of such concurrent instances of an activity. The ESTEREL model of the activity diagram shown in Fig. 6.5 is shown below. The module R is the coordinating module for A and B. In this model we assume that there could be at most two instances of activity B as shown by the two modules named B1 and B2 in the code. In Fig.6.5 the number shown in bracket indicates the maximum possible number of instances of activity B. Here we assume calling external tasks as final activities for ActivityNodes A and B.

```
module A :
    output InA ;
    return ExitA ;
    task activityA ( ) ( ) ; % external asynchronous task declaration
    exec activityA ( ) ( )  return ExitA% external action
      ||
    abort
          sustain InA ; % indicates module A is active
    when  ExitA
end module
```

```
module B :
    return ExitB ;
    output InB ;
    task activityB ( ) ( ) ; % external asynchronous task declaration
    exec activityB ( ) ( )  return ExitB% external action
      ||
    abort
          sustain InB ;
    when  ExitB
end module
module R :
    return ExitA,ExitB1,ExitB2 ;
    input InA, InB1,InB2 ;
    task activityA ( ) ( ) ; % external asynchronous task
    task activityB ( ) ( ) ; % external asynchronous task
    input start ;
    signal b1b2, free ∈
         loop
              await [  start  or ExitA  ];
              present  free  then [
                                        abort
                                              run  A
                                        when  ExitA
                   ]
              end
         end
          ||
         loop
              present [  not InB1  ] then % First instance of B
                   [
                        await  ExitA ;
                        % Signal renaming
                        run  B1/B  [ signal  ExitB1/ExitB,InB1/InB  ]
                   ]
                 else [  present  not InB2  then
                              [    % Second instance of B
                                   await  ExitA ;
                                   emit b1b2 ;
                                   %Signal renaming
                                   run  B2/B  [ signal  ExitB2/ExitB,InB2/InB  ]
                              ]
                            else [
                                   await [  ExitB1  or ExitB2  ];
```

```
                    emit start
                ]
            end
        ]
        end present
    end
      ||
    loop
        await start ;
        abort
            sustain free% free is on when B1 is active but B2 is dormant
        when b1b2
    end
  end
end module
```

Since each run B produces a separate instance of the task associated with the activity B, several simultaneous instances of activity associated with B can exist. In this case one should specify the number of instances of such activities. The model here shows capability of running two identical activities concurrently.

6.2.4 Modeling Exception

Fig. 6.6, shows the exception in an activity diagram. The node which is aborted due to the exception is called the protected node and the receiving node is the exception handler node. An exception handler is an element that specifies a body to execute in case the specified exception occurs during the execution of the protected node. In Fig. 6.6, Activity Node `ProcessOrder` is the protected node and `CancelOrder` is the exception handler and `CancelOrderEvent` is the exception input. This can be modeled in ESTEREL as shown below..

```
module B
  input cancelOrderEvent, ExitProcessOrder ;
  trap T in
    run ProcessOrder
      ||
    abort
      loop
        await cancelOrderEvent ; % Watch exception event
        exit T
      end
```

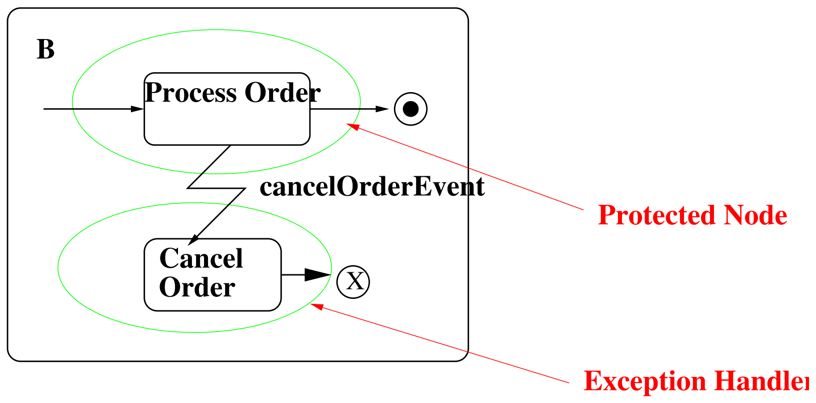


Figure 6.6: Exception Node

```
        when  ExitProcessOrder
      handle  T  do
        run  cancelOrder% Exception Handler
    end
  end
```

Here the activity `ProcessOrder` is preempted and the the activity `cancelOrder` is executed on raising the exception event `cancelOrderEvent`.

6.3 Activity with Data and Nesting

In many instances one *ActivityNode* may need to pass a data to another ActivityNode for processing by the *Activity* performed at that ActivityNode. For example if P and Q are two ActivityNodes and P is required send a data X to Q as shown in Fig.6.7 then this can be modeled using the mechanism shown below. The ExitS signal emitted by the activity node S is used for synchronizing the fact that the data token is available at the end of activity P.

```
module main
inputoutput X:type % X is the data which is passed between activities
    run P(X)
    await immediate exitP
    run Q(X)
end module
module P
output X:type
...
    emit ExitP
end module
```

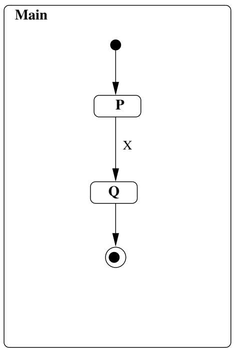


Figure 6.7: Object node with data

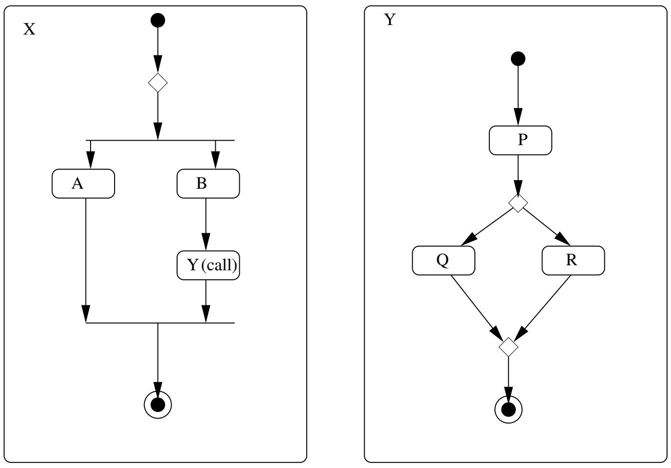


Figure 6.8: Activity with Nesting

```
module X            || module Y
...                 ||          ....
          run A     ||          run P;
                    ||          if e = u then
          run B;    ||                              run Q
          run Y     ||          else if e = v then
          ...       ||                              run R
          ...       ||          end
end                 || end
```

```
module Q
input X:type
task QActivity()(); % declaration of asynchronous task
...
     exec task QActivity(X) return ExitQActivity;
...
end module
```

In our model, Activity Diagrams with nested *call* can be modeled naturally. Let us assume that one activity Y is nested in another activity X as a `call Y` action in the activityNode C of X shown in Fig. 6.8. This can be modeled by using the *run Y* construct of ESTEREL . The following ESTEREL fragment describes the nested call of the Fig.6.8.

6.4 Communication in Activity Diagrams

The notion of communication between two Activity Diagrams can be nicely modeled in the Communicating Reactive Processes (CRP) [10] framework. The CRP model consists of network $\|_i^n M_i$ of ESTEREL modules, each having its own inputs and outputs and its own notion of instants. The network is asynchronous and the nodes communicate though synchronous channels. In this model, each M_i is an Activity Diagram each of which evolve locally with its own input and output and mutually independent notions of time [10]. Signals may be sent or received in activity diagrams through channels and is denoted by the common send and receive nodes. As an implementation model, one can think of an asynchronous layer (task) that handles rendezvous by providing the link between the asynchronous network events and node reactive events. The shared task can be called as channel. Fig. 6.9, shows a simple example of an activity diagram showing two component activities *PrintServer* and *PrintClient* communicating data (as files) through a channel. The CRP code for the same is shown below.

```
module PrintServer
input channel printq from PrintClient : FILE % CRP channel
......
    receive(printq,file) % send data file to printq
.....
end module
module PrintClient
output channel printq from PrintServer :FILE % CRP channel
...
    send(printq,file)    % receive data file from printq
....
end module
```

The send and receive [100] are communication primitives realizing the communication rendezvous between two locally synchronous programs. The primitive `send` blocks until sending data on the named channel succeeds and the primitive `receive` blocks until a communication succeeds on the named channel and the value assigned to the variable.

6.5 Simulation and Code Generation

Above we have shown how activity diagrams can be transformed into ESTEREL model. We are augmenting our previous work [18] to translate them automatically. The ESTEREL model can be simulated by using the *xes* interface. *Xes* is the simulator freely available along with the ESTEREL distribution. The simulator can be generated by compiling the ESTEREL program with the xes library. The simulation gives the user a clear picture of the execution of the activity diagrams and checking conformance to requirement is easy. We are also building simulators directly in the domain of input activity diagrams whereby one can see the simulation graphically.

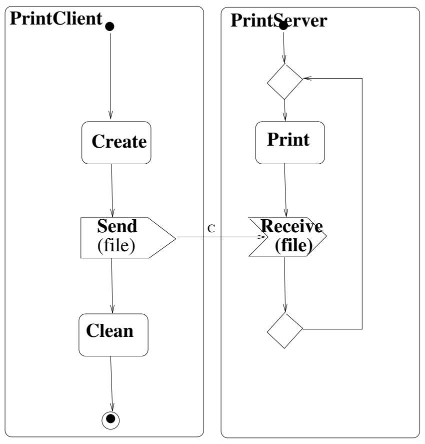


Figure 6.9: Object node with communication

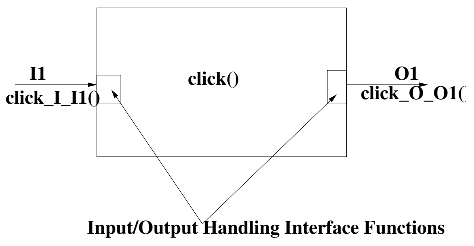


Figure 6.10: Activity to Code Mapping

6.5.1 Code Generation

There are two orthogonal levels of semantics, both indispensable: the intuitive level, where semantics must be natural and easy to understand, and the formal level, where the semantics is rigorously defined and fully non-ambiguous. Having formal semantics for the languages also makes code generators much easier to develop and verify. The translation process from Activity Diagrams to High Level Language (HLL) code like C is based upon sound proven algorithms that the ESTEREL code generators directly implement. By providing a formal semantics based on the synchronous paradigm and ESTEREL , it is easy to build correct code by construction, using ESTEREL -C/Java code generators. We assume ESTEREL -C code generator for further discussion.

For actual execution of the code , the generated code must also be linked with some extra layer of code that realizes the interface with the outside world which detects input events, read data and realizes output events and send data.If for example the module `click` should react to an input event, composed for example of one input tokens I1 as shown in Fig. 6.10. The sequence will include call to one automatically generated input C function `click_I_I1()` . This should be followed by call to the reaction function by executing the C code `click()`, followed by a call to output C function `click_O_O1()`.

The automatic code building process is achieved using the rules described above

1. Model the flow as an activity diagram model
2. Transform the model into the ESTEREL model following the rules as described above. These can be automated by encoding them in a model transforming algorithm similar to [18].
3. Describe interfaces as required by the ESTEREL modules regarding inputs and outputs.

4. The activities to be performed in the software `exec tasks` are to be encoded in the host language and operating systems.

6.6 Verification

We only discuss the verification approach in case of conventional UML AD (i.e. without compensation). The model captures the operational semantics of activity diagrams. However it is not amenable to formal verification using model checking due to presence of asynchronous tasks invoked by the `exec` statements. For the purpose of verification, it is required to do a control abstraction of the ESTEREL models whereby we only retain the labels where the task is to be created. The derived model is thus converted into a pure ESTEREL program and one can perform a constructive causality analysis using the ESTEREL compiler option of *causal*. This model can then be converted into an automaton in BLIF (Berkley Logical Interchange Format) format, which is accepted by the ESTEREL model checker *xeve*.

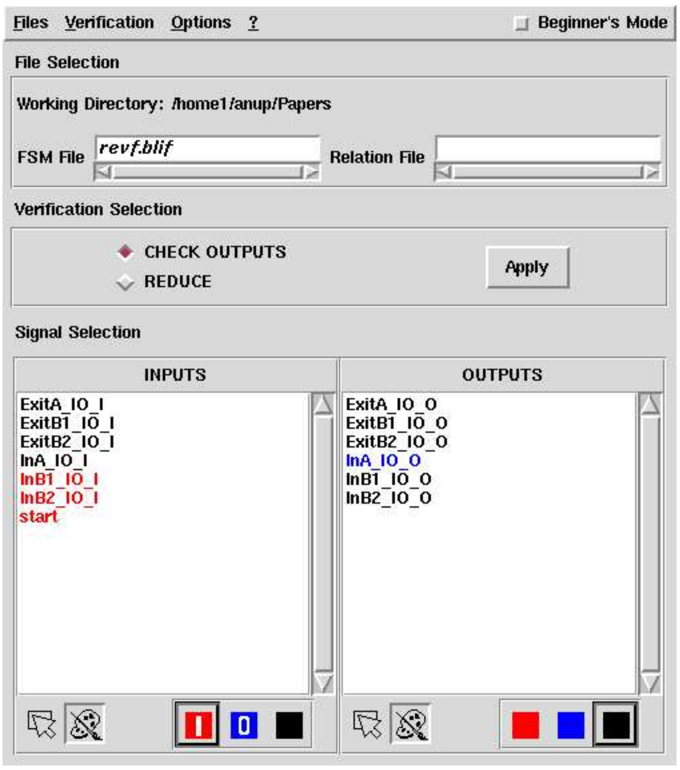
As an example, let us consider the activity diagram given in Fig. 6.5 with the following very simple safety property: *when both B1 and B2 activities are going on activity A cannot be started*. It is to be noted here that B1 and B2 are two incarnations of the activity B. This is assuming that there is no queuing of input. This could be verified by *xeve*. The screen shots taken from *xeve* are included here in Figs.6.11,6.12 for reference.

6.7 Implementation Model for Compensating Activities

The compensating activity diagrams can be represented as a model in a reactive framework based upon Mode Automata [83]. Mode Automata is a reactive language which combines synchronous data flows with running modes. The compensating activity could be considered as having two modes: normal and compensating modes. The normal mode defines the activity in the forward direction and the compensating mode defines the activity which is run in case of a failure in the subsequent activities. Fig. 6.13 shows two compensating activities PP and QQ. The modes of PP are also shown as a Mode Automaton in bottom of the the Fig.6.13. The compensating activities PP and QQ are shown as two concurrent state machines. In the forward mode of P the variable x is incremented by 1. In case the forward activity of $QQ = Q \div \bar{Q}$ fails, it is compensated by the compensating mode $\bar{P}$ of PP. The actual action in each activity is written as a dataflow equation in the box. These could be the *tasks* as shown in earlier in the ESTEREL code. In Fig. 6.14 we show the composite Mode Automaton.

6.8 Summary

In this chapter, we have explored the specification of operational semantics for the Activity Diagrams of UML 2.0 in a synchronous style. The semantics is good for simulation, code generation and verification. Our initial experience shows that verification of Activity Diagrams in this approach can be applied to moderately large examples. All the constructs can be expressed uniformly in the constructs of Esterel. In this approach the external action done in the activitynode can be easily modeled as an external task in the Esterel language. The exception handling in Petri Nets as shown in [106] is rather difficult which can be modeled easily in our framework.



File Selection

Working Directory: /home1/anup/Papers

FSM File *revf.blif*

Relation File

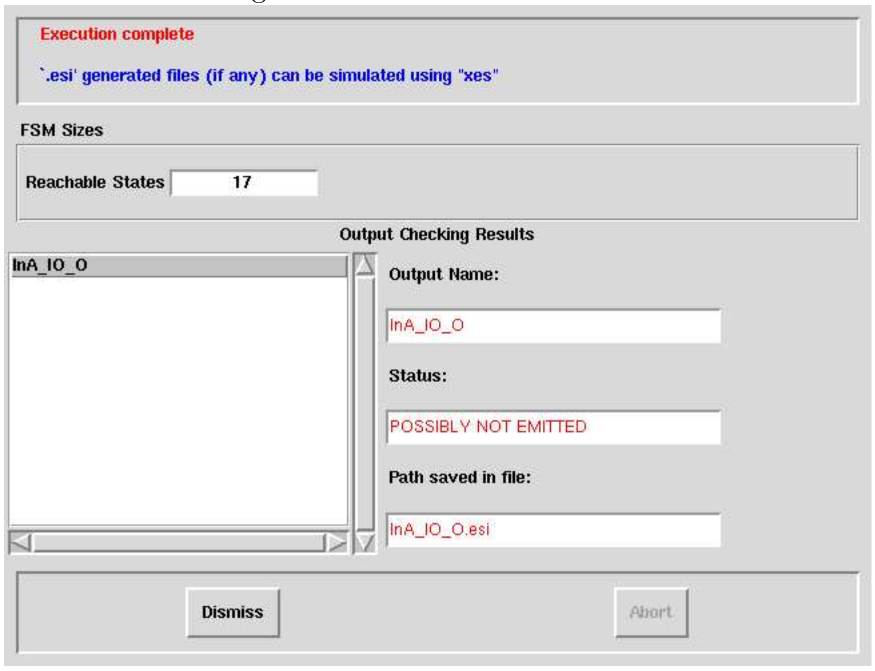
Verification Selection

[✓ CHECK OUTPUTS] REDUCE

Signal Selection

INPUTS	OUTPUTS
ExitA_IO_I	ExitA_IO_O
ExitB1_IO_I	ExitB1_IO_O
ExitB2_IO_I	ExitB2_IO_O
InA_IO_I	InA_IO_O
InB1_IO_I	InB1_IO_O
InB2_IO_I	InB2_IO_O
start	

Figure 6.11: Verification Screen



Execution complete

`.esi' generated files (if any) can be simulated using "xes"

FSM Sizes

Reachable States 17

Output Checking Results

InA_IO_O

Output Name: InA_IO_O

Status: POSSIBLY NOT EMITTED

Path saved in file: InA_IO_O.esi

Figure 6.12: Output of Verification

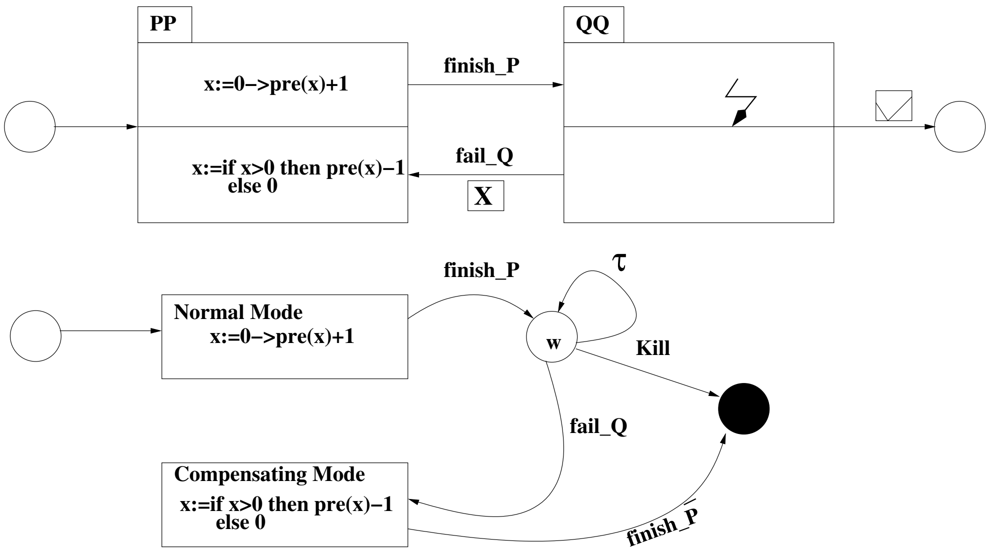


Figure 6.13: Mode Automata for Activity with Compensation

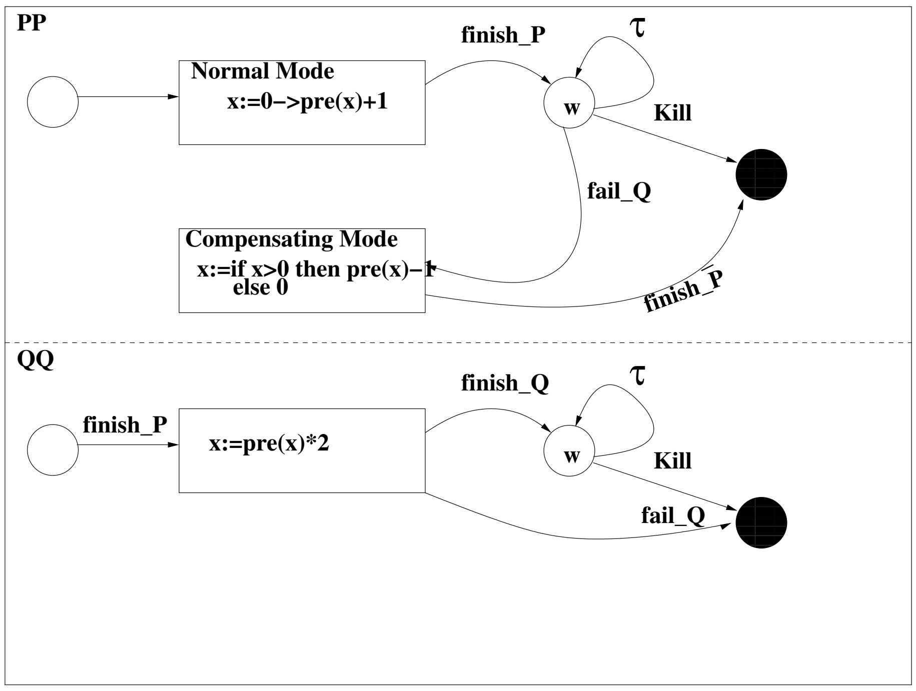


Figure 6.14: Composition of Mode Automata

Chapter 7

A Specification Language for Choreography in Distributed Systems

7.1 Introduction

A distributed system is a collection of computers that are spatially separated and do not share a common memory. The processes executing on these computers communicate with one another by exchanging messages over communication channels. Designing distributed systems is a complex process because of asynchrony, limited local knowledge and failures. An example of distributed computing is a collection of computing entities connected over a network which interact through well defined interfaces like TCP/IP, CORBA or some other middleware.

Service oriented computing [104] is a new emerging paradigm for distributed computing. Services are autonomous computational entities offering services and web services are the most common application of service oriented computing. The terms *orchestration* and *choreography* are used to define two different flavors of service oriented computing. While orchestration is used to describe a single view point model, choreography is about specifying the service orchestration in a global model. Choreographs define the sequence of exchanging messages between two (or more) independent participants or processes by describing how they should cooperate. The main advantage of a global definition approach is that it separates the process being followed by an individual system within a *domain of control* from the definition of the sequence in which each system exchanges information with others as illustrated in Fig.7.1. This means that, as long as the *observable* sequence does not change, the rules and logic followed within the domain of control can change [84].

The specification, design and implementation of service oriented applications need to address at least the following three major aspects:

- Orchestration of Services: Provide the specification and realization of processes that drive the message exchanges among web services in the context of arbitrary delays and failures.

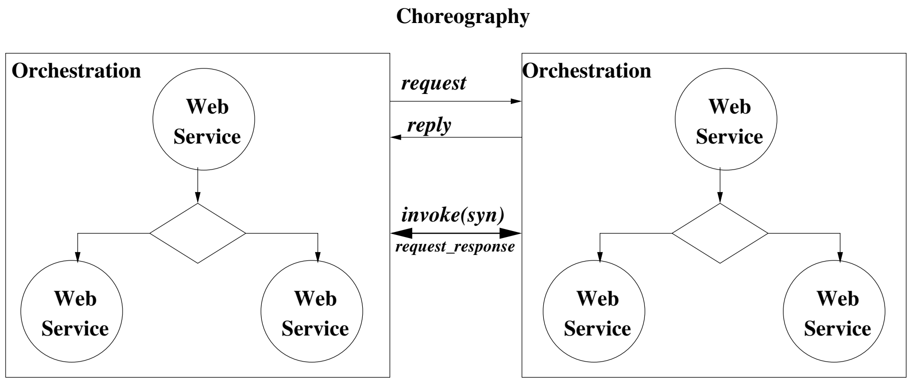


Figure 7.1: Orchestration+Choreography

- Conversation: Specification of request-response patterns around web services.
- Choreography: Specification of collective message exchanges among interacting web services providing a global, message-oriented view with observations and controls.

In this chapter, we focus on structuring choreography as a set of message exchanges among various participating roles as conversations. Our main contribution in this paper is a language *ScriptOrc*, which integrates orchestration with scripting to abstract conversations leading to an effective modular specification of service choreography. In our approach, we are using the power of Scripts [47] and Orc [86] and capture the conversations that form the key factor for web service abstractions. Our language is based on a reactive framework [10]. As pointed out in [86], a reactive semantics for web services will provide a good basis for static and dynamic resource discovery and exploitation as it leads to good dynamic monitoring schemes. We shall demonstrate the formalism with an example from workflow.

The chapter is organized as follows: section 7.2 provides a brief introduction to the issues in the process specification. The role of Scripts in abstracting conversations as patterns of communication is discussed in section 7.3. The formal language for actual specification is introduced in section 7.4 and in section 7.5, we demonstrate the use of the language with an example. Finally, we summarize and conclude in section 7.6.

7.2 Background

A choreography defines collaborations among interacting participants. It can be structured as a container for a collection of activities to be performed by the participants. There are two types of activities identified in [111]: *Basic activities* and *Control-flow activities*. Basic activities include a SKIP action, which does not do anything; an assign activity, which assigns, within one role, the value of one variable or an expression to another variable; and an interaction activity, which results in an exchange of information between participant roles and possible synchronization of their observable information changes and the actual values of the exchanged information. An interaction activity consists of:

- participant roles in realizing the objective; in web services, tasks are often understood as some kind of registration needed due to security and other requirements;
- exchange of information along with direction(s);
- observable information changes;
- operation performed by the recipient.

The type of interactions is described by the possible actions on the communication channel, which falls into three types: request, response, or request-response with respect to the agents' role in accomplishing the task; in a sense, the *role* provides either an implicit or explicit specification of computation. Note that the patterns of communication can be recursive. Apart from the requirements of handling multi-party interactions and allowing modularity and composability , the other goals of a choreographic language [113] are:

- Information Driven: Choreography describes the way participants take part and maintain the locus by recording the state changes caused by exchanges of information and their reactions.
- Information Alignment: Choreography allows the participants to communicate and synchronize their states and the information they share.
- Exception Handling: Choreography needs to define handling of exceptions.

7.3 Modeling conversations as patterns of communication among processes

In services oriented architecture of distributed systems, patterns of communications among agents or processes follow a clear role. For instance, when a customer is requesting a travel agent for air-ticket between destinations A and B, we can see two roles: (i) the customer looking for possible itineraries between A and B possibly with some constraints such as cost, single hop etc, and (ii) the travel-agent with the task of providing answers to the specified queries from the customer. Often, the request-response looks like a database query even though it is not necessary that the structure of results come from one static database; quite often it could be an aggregation among several databases which are often dynamic realized through a protocol among the communicating processes.

We use the abstraction referred to as, Script in [47], in the context of distributed computing to provide a good abstraction for web-service conversations. We define Scripts consisting of:

- *Roles* are formal process parameters of goal-oriented task abstractions in which actual processes enroll. Each role has a set of parameters that consists of data parameters as well as role parameters. Note that the same process can be in different roles for different scripts (note often the role could be symmetrical).
- *Data Parameters* are the classical parameters for binding actual parameters with names.
- *Body* is the executable abstraction in terms of roles that would be enrolled by processes. The body essentially describes the behaviour articulating the actions that will be performed by the roles jointly or other wise.

Scripts have attributes like

- *delayed initiation*: tasks does not start immediately; reason could be due to await for other processes to enroll or wait for time. For instance, the classical rendezvous where one needs to wait for the dual partner to do the task. In general, one could specify a threshold for the script to proceed.
- *delayed termination*: All the roles that have finished their action also need to wait for all the roles participating to terminate (E.g.,broadcast message delivery confirmation).
- *immediate initiation*: start immediately (E.g., asynchronous message transmission).
- and *immediate termination*. The role terminates as soon as its' role is complete (E.g., asynchronous task creation).

The importance of roles and scripts, is highlighted using the following example from [86]. An office assistant (Bob) is asked to arrange the visit of a Speaker (Alice) for which he contacts the speaker, requesting for a set of possible dates. The speaker responds by choosing a possible dates for the visit. The assistant then contacts hotel and airline reservation sites with the dates. He awaits confirmation from the hotel and airline reservation. Once he receives the confirmation, the assistant makes the choice and confirms to the hotel and airlines. The entire conversation is choreographed under the role of the office assistant. We describe a script below to describe possible conversations considering the fact the Office Assistant performing the complete choreography.

```
SCRIPT1:
INITIATION IMMEDIATE
TERMINATION DELAYED
  Bob enrolls as OFFICE ASSISTANT
  Alice enrolls as SPEAKER
  Bob sends request to Alice with a set of dates
  Alice replies with a date
----------------------
  SCRIPT2:
```

```
  INITIATION IMMEDIATE
  TERMINATION DELAYED
    HILTON enrolls as HOTEL
    TAJ enrolls as HOTEL
    [Bob sends reservation request to HILTON
    HILTON replies to Bob with cost]
      |
    [Bob sends reservation request to TAJ
    TAJ replies with cost]
    Bob selects a Hotel ∈ { HILTON,TAJ }
      with lowest cost
  END
-----------------------
  SCRIPT3:
  INITIATION IMMEDIATE
  TERMINATION DELAYED
    DELTA enrolls as AIRLINES
    UNITED enrolls as AIRLINES
    [Bob sends reservation request to DELTA
    DELTA replies with flight no and cost]
      |
    [Bob sends reservation request to UNITED
    UNITED replies with flight no and cost]
    Bob selects a Flight ∈ { DELTA, UNITED}
      with lowest cost
  END
-----------------------
  Bob sends (Hotel, Flight) info to Alice
  Alice confirms to Bob
  Bob confirms to Hotel
    |
  Bob confirms to Airlines
END
```

Thus, we can identify the roles as *Office Assistant*, *Speaker* and *Hotel* and *Airlines*. The roles Hotel and Airlines have *multiple instances*. The initiation of the scripts could begin the moment stake holder roles have enrolled themselves in the script. Thus the first script can start the moment *Office Assistant* and *Speaker* have enrolled in the roles. The Scripts SCRIPT2 and SCRIPT3 can start the moment the Hotels sites and Airlines have enrolled themselves in the scripts which could be initiated in parallel. We can see the *request, request-response* patterns of communication in the above script. Completion of SCRIPT1 depends on values published by SCRIPT2 and SCRIPT3 which is line with asynchronous parallelism of Orc [86].

In another scenario, the office assistant selects a hotel and flight through a hotel reservation portal and a flight reservation portal. The office assistant has to provide the *constraint like cost and a suitable time of departure*. The hotel reservation site involves an orchestration of many hotels (shown in thick lines) whose roles are registered with the site. Similarly the airline reservation site orchestrates between various airlines which are registered with it. In this case, we can note that the master choreographer *Office Assistant* is initiating a conversation in each of the portals to give him a suitable hotel and flight meeting the constraints. In this case we can identify three distinct roles namely *Office Assistant*, *Speaker*, *Hotel Reservation Portal* and *Flight reservation Portal*. The *Hotel Reservation Portal* again depends upon the various hotels which register themselves. The same is with *Flight reservation Portal* and *Airlines*. The Hotel Reservation and Flight reservation scripts could be initiated in parallel. Scripts similar to the one shown above can be written for this case.

7.4 ScriptOrc: A Language Abstraction for Choreography

ScriptOrc is based on a reactive framework [10] and provides constructs to describe conversations and preemption of conversations based on timeouts.

The basic choreography is defined in terms of a control structure around a set of basic activities, locations and variables. The set of basic activities define expression evaluation and assignments to variables in each site participating in a role. Composition of basic activities constitute the kernel command structures. Control Structures group Basic Activities and other Control Structures together in a nested structure to express the logic and decision flow involved in the Choreography.

Notation:

- $\tilde{x} = \{x_1, x_2, \ldots x_n\}$ denotes tuples of variables.
- α,β denote events.
  - An event could be a notification like completion of a conversation, exceptions (communication/computation) or an external interrupt due to cancellation.
- σ denotes state and is the valuation of state variables.
- $\mathcal{L}$ denotes a set of distinct locations like, say AMEX, Travelex, NDTV etc (denoted by $r,\ r_1,\ r_2$).
- $\mathcal{A}$ denotes a set of basic activities understood by the various sites/locations/agents.

The basic choreography can be realized as a set of conversations where conversations corresponds to interactions with other agents or local computations. Denoting C, C_1 and C_2 for conversations, then constructs defined in Table 1, can be used to define and compose the

C	$::=\ \mathbf{0}$	SKIP
	$\mid x@r := e$	ASS
	$\mid x@r := e$ when y	T-ASS
	$\mid req(\vec{\rho}_{A_1}[k_1 \in \mathcal{N} : criticalset], \cdots \vec{\rho}_{A_n}[k_n \in \mathcal{N} : criticalset], \tilde{x_1}, \cdots, \tilde{x_n}, C)$	REQ
	$\mid reqres(\vec{\rho}_{A_1}[k \in \mathcal{N} : criticalset], \cdots \vec{\rho}_{A_n}[k \in \mathcal{N} : criticalset],$ $\tau, \tilde{x_1}, \cdots \tilde{x_n}, \tilde{z_1}, \cdots \tilde{z_n}, C)$	RQR
	$\mid C_1;\ C_2$	SEQ
	$\mid C_1 \mid C_2$	PAR
	$\mid C_1 \lhd b \rhd C_2$	ALT
	$\mid b * C$	LOOP
	$\mid$ abort C_1 when α handle C_2 TIMEOUT(τ) C_3 end	ABT

Table 7.1: Syntax of ScriptOrc

conversational aspect of choreography. Note, even scripts can be treated as conversations in that sense.

The basic kernel constructs for request/response are:

- $req(\vec{\rho}_{A_1}[k_1 \in \mathcal{N} : criticalset], \cdots \vec{\rho}_{A_n}[k_n \in \mathcal{N} : criticalset], \tilde{x_1}, \cdots, \tilde{x_n}, C)$: This is a *request for one-way conversation* consisting of:
  - $\rho_{A_1}, \cdots, \rho_{A_n}$ denote roles interacting via the data parameters $\tilde{x_1}, \cdots \tilde{x_n}$.
  - $k_1, \cdots k_n \in \mathcal{N}$: minimum number of enrollments needed in script C before it can start.
  - Roles are bound to script C.
- $reqres(\vec{\rho}_{A_1}[k_1 \in \mathcal{N} : criticalset], \vec{\rho}_{A_n}[k_n \in \mathcal{N} : criticalset],$ $\tau, \tilde{x_1}, \cdots \tilde{x_n}, \tilde{z_1}, \cdots \tilde{z_n}, C)$. This is a request response conversation (two way or multi-party) where
  - Z_is are the response results, and
  - all other parameters are the same as in *req*.
  - Note that, as we are having `timeouts`, we assume that request and response cannot happen instantaneously.
- The command `abort` C_1 `when` α `handle` C_2 `TIMEOUT`(τ) C_3 `end` denotes a conversation preempts a conversation if the needed response does not arrive within the timeout τ or is preempted by an event α in the environment. Note that the timeout is specified with reference to the process that is hosting and executing the conversation; it could be the local clock or a global clock in a perfectly synchronous system.
- The conversations follow on classical lines; note that SKIP amounts to no action except for the enrollment of processes 9similar to pure synchronization).

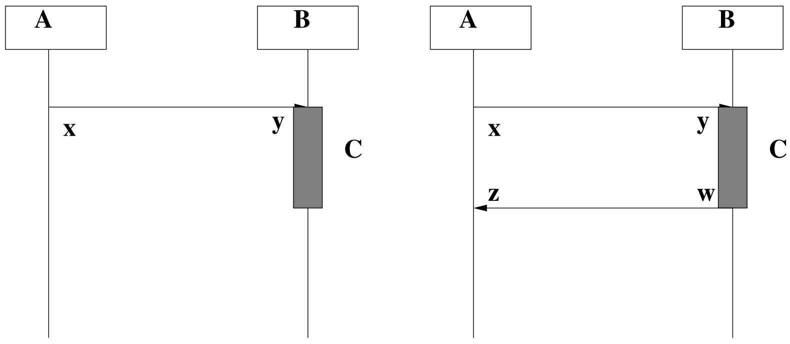


Figure 7.2: Request and Request-Response Operations

The primitive $req(\rho_A, \rho_B, \tilde{x}, \tilde{y}, C)$ is an invocation of the conversation C in the context of role ρ_B initiated by role A as illustrated in Fig. 7.2 (later denoted by $C^{\rho_A}[\rho_B]$ but the context of agent ρ_A is omitted). Let $r_1 \in \mathcal{L}$ and $r_2 \in \mathcal{L}$ be two sites in roles ρ_A and ρ_B, and $m ::= (r_1, r_2, \tilde{x}, \tilde{y}, dir)$ where $dir = \{\uparrow, \downarrow\}$ indicates whether it is a request or response for m. If $\mathcal{M}$ denotes the underlying semantics (not going into formalisms due to lack of space) then

$$\mathcal{M}[\![\langle \rho_A, \rho_B, o, \tilde{x}, \tilde{y}, \uparrow \rangle]\!] \triangleq \tilde{x}@r_1 \xrightarrow{send} \tilde{y}@r_2$$

This captures the binding of various parameters as described above of the roles, data etc. Note that, as mentioned already, there shall be a observable time difference between $\tilde{x}@r_1$ and $\tilde{y}@r_2$ when r_1 and r_2 are distinct.

$$\mathcal{M}[\![req(\rho_A, \rho_B, \tilde{x}, \tilde{y}, C)]\!] \triangleq \mathcal{M}[\![\langle \rho_A, \rho_B, o, \tilde{x}, \tilde{y}, \uparrow \rangle]\!];\ \mathcal{M}[\![C[\rho_B]]\!](\tilde{y})$$

This corresponds to transmission of parameters; followed by actions for the request/response.

Note that *req* or *reqres* conversation may not start unless all participants in the conversation have enrolled in their respective roles. In that sense, the *initiation* and *termination* of a conversation are always delayed.

The primitive *reqres* is used to describe the invocation of the request-response operation as illustrated in Fig. 7.2. The semantics for *reqres* is an invocation of the conversation C in role ρ_B, performed by role ρ_A. The parameter τ denotes the timeout of response and if τ=0 then the command waits for until a response arrives. The conversation $\mathbf{C}$ in *reqres* which can invoke other conversations happens between *request* and *response*. The conversation $\mathbf{C}$ in *reqres* could also be an orchestration service. Semantically, this can be defined as given below:

$$\begin{aligned}\mathcal{M}[\![reqres(\rho_A, \rho_B, \tilde{x}, \tilde{y}, \tilde{z}, \tilde{w}, C)]\!] \ \triangleq\ & \mathcal{M}[\![\langle \rho_A, \rho_B, o, \tilde{x}, \tilde{y}, \uparrow \rangle]\!]; \\ & \mathcal{M}[\![C[\rho_B]]\!](\tilde{y}); \\ & \mathcal{M}[\![\langle \rho_B, \rho_A, o, \tilde{w}, \tilde{z}, \downarrow \rangle]\!]\end{aligned}$$

The meaning of other constructs are given below:

- $\mathbf{0}$ denotes the Null process that terminates instantaneously.
- Sequence $(C_1;\ C_2)$: denotes the sequencing of conversations – C_2 starts after the completion of C_1.
- Choice $(C_1 \lhd b \rhd C_2)$: denotes choice of conversations based on either local or global (environment) conditions.
- Parallel $(C_1 \mid C_2)$: denotes concurrent execution and terminates when both of them terminate.

7.4.1 Operational Semantics of ScriptOrc

Here, we provide a basic operational semantics of ScriptOrc using reactive style. As we use explicit preemption/timeout, we use a global clock to see the progress of time the participating precesses or agents.

$$\text{ASS}\frac{\langle \sigma(e) = v \rangle}{\langle x@r := e, \sigma \rangle \rightarrow \langle \mathbf{0}, \sigma[x@r \mapsto v] \rangle}$$
$$\text{T-ASS}\frac{\langle \sigma(e) = v \rangle}{\langle x@r := e \text{ when } y, \sigma[y = \top] \rangle \rightarrow \langle \mathbf{0}, \sigma[\mathbf{x@r} \mapsto \mathbf{v}] \rangle}$$

$$\text{SEQ}\frac{\langle C_1, \sigma \rangle \xrightarrow{\alpha} \langle C_1', \sigma' \rangle}{\langle C_1;\ C_2, \sigma \rangle \xrightarrow{\alpha} \langle C_1';\ C_2, \sigma' \rangle}$$
$$\text{SEQ}\frac{\langle C_2, \sigma \rangle \xrightarrow{\beta} \langle C_2', \sigma' \rangle}{\langle SKIP;\ C_2, \sigma \rangle \xrightarrow{\beta} \langle C_2', \sigma' \rangle}$$
$$\text{ALT}\frac{}{\langle C_1 \lhd b \rhd C_2, \sigma \rangle \xrightarrow{b} \langle C_1, \sigma' \rangle}$$
$$\text{ALT}\frac{}{\langle C_1 \lhd b \rhd C_2, \sigma \rangle \xrightarrow{\neg b} \langle C_2, \sigma' \rangle}$$

The rule for site enrollment is based upon willingness of a site to enter into a role with an empty slot in the critical-set. A conversation can start at a site only when it has a an available roll in the conversation. The following rules provide semantics for role enrollment and concurrent conversations.

$$\text{Enrolling}\frac{r_i \in \mathcal{L}, k_i < \mathit{criticalset}(A) \langle r_i, A \rangle \mapsto \rho_A}{\langle \rho_A, C_1, \sigma \rangle \rightarrow \langle C_1[\rho_A], \sigma' \rangle}$$

PAR-CI Concurrent interaction of independent conversations

$$\frac{\langle C_1, \sigma\rangle \xrightarrow{\alpha} \langle C_1', \sigma'\rangle}{\langle C_1 \mid C_2, \sigma\rangle \xrightarrow{\beta} \langle C_1' \mid C_2, \sigma'\rangle}$$

Similarly the dual rule for C_2

PAR-CT Parallel construct terminates when both terminate

$$\frac{\langle C_1, \sigma\rangle \xrightarrow{\alpha} \langle \mathbf{0}, \sigma'\rangle, \langle C_2, \sigma\rangle \xrightarrow{\beta} \langle \mathbf{0}, \sigma'\rangle}{\langle C_1 \mid C_2, \sigma\rangle \xrightarrow{\alpha\cup\beta} \langle \mathbf{0}, \sigma'\rangle}$$

The rule `PAR-N-SYNC` denotes interacting (synchronizing) conversations. In `PAR-N-SYNC` synchronizing event between the conversations C_1, C_2 is represented by $\bowtie (C_1, C_2)$. Let us assume two sites $r_1, r_2 \in \mathcal{L}$ that have enrolled into roles ρ_A and ρ_B. Since conversations are essentially programs, $\bowtie$ can be viewed as a parallel composition of labelled transition systems for C_1 and C_2 that synchronize, that is, there is proper matching of req and $response$ at all points in order. If $\ell_1,\ \ell_2$ denote actions in $First(C_1)$ and $First(C_2)$, $match(\ell_1, \ell_2)$ denotes matching of the req-response pairs then

$$\bowtie (C1, C_2) \triangleq \exists \ell_1 \in First(C_1), \ell_2 \in First(C_2) \bullet \sigma \models match\langle \ell_1, \ell_2\rangle \wedge \bowtie (rest(C_1), rest(C_2))$$

PAR-N-SYNC

$$\frac{\langle \rho_A, C_1, \sigma\rangle {\rightarrow} \langle C_1[\rho_A], \sigma'\rangle, \langle \rho_B, C_2, \sigma\rangle {\rightarrow} \langle C_2[\rho_B], \sigma'\rangle}{\langle C_1[\rho_A]\ ,\ C_2[\rho_B], \sigma\rangle \xrightarrow{\bowtie(C_1,C_2)} \langle C1'[\rho_A]\ ,\ C2'[\rho_B], \sigma'\rangle}$$

ABORT (ABT)

$$\frac{C_1 \xrightarrow{*} SKIP}{\langle \text{abort } C_1 \text{when } \alpha \text{ handle } C_2 \text{ TIMEOUT}(\tau) C_3 \text{ end ; } C_4, \sigma\rangle \xrightarrow{\beta}}$$

The conversation can evolve in four ways based on environment

$$(1)\frac{}{\langle C_4, \sigma\rangle}$$

$$(2)\frac{}{\langle \text{abort } {C_1}^{\beta} \text{when } \text{ handle } C_2 \text{ TIMEOUT}(\tau) C_3 \text{ end ; } C_2, \sigma'\rangle} \beta \neq \alpha$$

$$(3)\frac{}{\langle C_2;\ C_4, \sigma\rangle} \beta = \alpha$$

$$(4)\frac{}{\langle C_3;\ C_4, \sigma\rangle} TIMEOUT(\tau) \in ENV \prec \alpha$$

Note that in an *abort* statement the *handle and timeout* segments are optional and is the primary reactive construct that allows preemption capable of handling and reacting to changes in the environment. Briefly,

(1) An abort statement terminates when the enclosed conversation itself terminates; C_4 denotes the conversation after the abort statement - it can be reached either through preemption (seeing α or normal termination of C_1 within the timeout statement.

(2) It continues to execute C_1 in the environment (Env) where the event β occurs but the conversation can only react to (or watching) α.

(3) If in the environment the event α occurs, then the abort statement terminates and starts the execution of next conversation after executing the event handler.

(4) However, if the timeout event occurs before the event $alpha$ and the statement C_1 has not yet terminated, then the timeout handler C_3 is executed followed by C_2.

A simplified syntax of the statement can be *abort C when α handle C_h TIMEOUT(τ) end*, where the conversation C is preempted by a local TIMEOUT event. The TIMEOUT event can occur only with respect to the local clock. Each process participating in the choreography is locally reactive [10] driving the complex operation over the asynchronous network. An execution of a set of processes $\{M_i \mid i \in I\}$ where I is the input event set can be defined as the set of (Input-output) events for the processes.

7.4.2 Clock Expressions in ScriptOrc

In the synchronous data flow model, each variable can be characterized both by its stream of values and by its global clock. A process transforms an input clock to output clock. This can be stated in terms of clock signatures that are relative to the appropriate clock variables. We can think of expression e in ScriptOrc as constant streams, variables x (bound during the course of execution), or stream variables like e `when xclk`. An assignment does not change the clock of a variable but sampling operators like *y:= e when x* defines the clock of y wrt x, where x could be assigned by a conversation. We are working on defining a more rigorous rules in ScriptOrc based on clock calculus.

7.5 Choreography of Speaker, hotel and Flight

The choreography defined in section 7.3, illustrating the choreography of Speaker, Hotel and Airline reservation performed by the Office Assistant could be scripted in our language as (variables with subscript *lc* are temporary variables) illustrated below. The subscripts SP and

OA denote information for roles Speaker and Office Assistant respectively.

```
SCRIPT : GetDate
INITIATION IMMEDIATE;
TERMINATION DELAYED;
ROLE $SPEAKER, OA$;
VAR $\tilde{date}_{OA}, \tilde{date}_{SP}$;
BEGIN
VAR $\tilde{w}_{lc}$;
#The vector $\tilde{w}_{lc}$ contains the set of dates
 $reqres(OA, SPEAKER, \tilde{date}_{OA}, \tilde{w}_{lc}, \tilde{date}_{SP}, d, C)$;
END
```

The script for hotel reservation and flight booking is given below. The function *Filter* does the computation for selection, meeting the constraints but details are not shown due to lack of space.

```
SCRIPT : GetHotel&Flight
INITIATION IMMEDIATE;
TERMINATION DELAYED;
BEGIN
VAR $\tilde{x}_{lc}, \tilde{y}_{lc}$;
ROLE $Speaker, OA, Hotel[n], Airline[m]$
 $reqres(OA, Hotel[n], \tilde{Pref}_{OA}, \tilde{x}_{lc}, \tilde{hotel}_{lc}, \tilde{hotel}, SKIP)$
 |
 $reqres(OA, Airline[m], \tilde{Pref}_{OA}, \tilde{y}_{lc}, \tilde{flight}_{lc}, \tilde{flight}, SKIP)$
 $hotel@OA = Filter(\tilde{hotel}, cost)$;
 $flight@OA = Filter(\tilde{flight}, cost, time)$;
 $\tilde{hf}_{OA}@OA = \langle hotel, flight \rangle$;
 $req(OA, Speaker, \tilde{hf}_{OA}, \tilde{hf}_{SP}, SKIP)$
 $req(OA, Hotel, confirm, SKIP)$;
 |
 $req(OA, Airline, confirm, SKIP)$;
END
```

We find that the Office Assistant (OA) is the master choreographer and it generates a complete arrangement for the Speaker's lecture date, his accommodation and flight, by performing a choreography between Speaker, Hotels and Airlines. In the above script, the communication requirements and control between various roles have been captured. The actual implementation (we show only for the function Arrange&Visit) now can be specified in Orc as

$$
\begin{aligned}
&ArrangeVisit(p, s) \triangleq \\
&\qquad GetDate(p, s) > d > \\
&\qquad (let(h, a) \text{ where } h :\in GetHotel(d), \\
&\qquad a :\in GetAirline(d)) \\
&\qquad > (h, a) > \\
&\qquad Ack(p, (h, a)) > q > \\
&\qquad (let(x, y) \text{where } x :\in Confirm(h), y :\in Confirm(a))
\end{aligned}
$$

7.6 Summary

In this chapter, we have demonstrated that modeling conversation with roles as in Scripts proposed in [47] leads to smooth Choreography abstraction. While Orc [86] is an elegant abstraction for Orchestration, ScriptOrc proposed in this paper provides a clean denotation of choreography and semantics follows on the lines of reactive semantics using techniques proposed for CRP in [10] with appropriate notions of events and states. In fact, we can use the Orc operators for our conversations in ScriptOrc with advantage.

Part II

Type Correctness and Translation Validation of Model Generated Code

Chapter 8

Model Transformation: Type Correctness of High Level Language Code

8.1 Introduction

Embedded software-based control systems are commonly constructed using model-based design environments. These environments allow the system designer to establish critical properties ensuring the reliability of the system directly at the model level, using a rich mathematical toolset. However, the software implementation substantially transforms the mathematical model by introducing numerous programming artifacts (aggregate data structures, pointers), arithmetic with finite representation and altering the numerical representation (platform-dependent floating/fixed-point arithmetic etc. Verifying that the safety properties of the system are preserved by the implementation is extremely challenging, yet in many cases critically important. Model based design environments usually come with an automatic code generator (autocoder) which automatically synthesizes an implementation of the embedded controller from the specification of its model. Automatic code generators are getting increasingly used in practical applications for they greatly simplify the implementation process. In industry concerned with development of systems for safety-critical systems however, auto code generation is used sparingly, because of use of complex algorithms, the generated code is considered to be not adequately trustworthy. However after constructing a verified model, it is always beneficial to use an auto code generator to keep the model and code in sync.

Static program analysis tools have recently proven successful in tackling the certification of embedded software-based control systems. Ensuring the absence of execution errors in software used in safety-critical systems is extremely important as the failure of such software can lead to system failure causing a loss of safety function. *Run Time Errors (RTEs)* arising out of poor *type system* (e.g. C language) are the most subtle but crucial type of errors found in software leading to system failures. Examples of such errors are array indices out of bound, divide by zero and arithmetic overflows/underflows etc. In this chapter we describes our work in developing techniques for detection of such type violations (also called runtime errors) using a deductive approach. Detecting run time errors statically through program analysis, helps in reducing efforts in activities like Debugging, White box testing and Code reviews. The other advantage is in reducing runtime overheads of checking program variables being restricted to its safe ranges during the execution of the program. Further once the type safety of a model generated code is ensured, it is possible to remove all run time checks to detect such errors thereby increasing the efficiency of execution.

In this chapter, we discuss an approach for detecting run time errors based on Type System [27] The technique is based upon modeling C programs as state transition systems encoded in the specification language of Prototype Verification System (PVS) [35]. *State Transition System* model of a program describes how variables in the program are modified as program executes from beginning to end. The description of a given program as state transition system in PVS language is called *PVS Specification* or *PVS Model* of the program. Since the specification language of PVS is strongly typed, the possible runtime errors in the C program result into type inconsistencies in the PVS specification. When the PVS specification is typechecked, these type inconsistencies automatically generate proof obligations called Type Correctness Conditions (TCCs). The PVS prover commands can be used for discharging the proofs of these TCCs. If all the TCCs are proved, the program can be declared as free of type violations and therefore corresponding runtime errors. Presence of any unproved TCC indicates that the program is not typesafe and it can be traced back to a possible runtime error in the C program. This is a deductive technique and requires user interaction but can handle infinite state systems. The proof process is automated to some extent using PVS strategies as scripts. However this approach of checking properties cannot be fully automated as it sometimes requires powerful theorem proving techniques that can be applied only interactively. Although the technique can be applied to general class of C programs, it is specifically targeted towards safety-critical software developed following software engineering practices like controlling module sizing, complexity(McCabe) etc.

The chapter is organized as follows. Section 8.2 gives an overview of the method. In section 8.3, we describe how C programs are modeled in PVS. A brief account on how TCCs are proved in PVS and how to locate the RTE is given in section 8.4. Section 8.5 presents a small example to illustrate the method. Section 8.6 summarises our experience and provides some discussion related to future work.

8.2 Type Inferencing Method

The method is based on modeling the C programs as state transition systems in PVS specification language. The states are encoded as the evaluation of the program variables over the data domains. The transitions are the execution of statements and modeled as the effect of modifying the set of state variables participating in the statement. Each state is expressed as a function of the previous state and thereby the entire program is encoded as a list of states

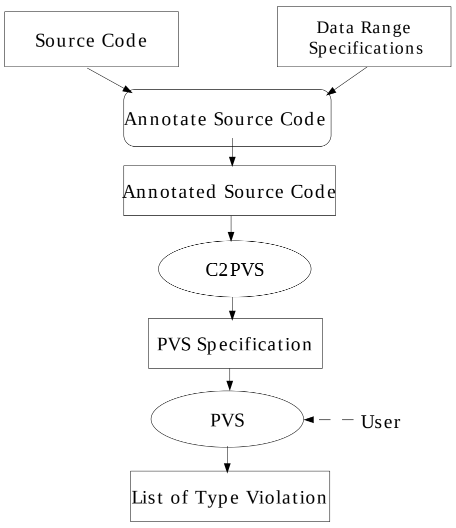


Figure 8.1: Process of Type Checking C Programs

each in terms of the previous states.

The datatypes of C (which are represented in the machine in finite size) are modeled in PVS as subtypes of PVS types with limited size. For example *int* datatype of C is represented as *restr_int* which is a subtype of *int* datatype of PVS restricted between the integer maximum and integer minimum. Each operator in C is modeled as a PVS function with operands having subtypes and return value having PVS type. For example, the binary + operator on *int* datatype of C is represented as a PVS function *add_restr_int* with *restr_int* operands and *int* return value. When + operator is used in any expression in the C program, it is replaced by the function *add_restr_int* in the PVS model. This modeling scheme causes the typechecker to generate TCCs for all possible runtime errors. This is explained in detail in section 8.3.

Fig. 8.1, shows the steps that are followed in carrying out runtime error detection. The source code is annotated with formal annotations extracted from the constraints on the input data of the program. These constraints are referred in the Fig. 8.1 as *Data Range Specifications*. The translator *c2pvs*, developed as part of this work, translates the C program to a *PVS specification* which is loaded in the PVS theorem prover. The type inconsistencies in the PVS specification are detected by the PVS typechecker causing generation of TCCs (Type Correctness Conditions), which need to be proved before the specification can be considered typesafe. An unprovable TCC shows that the program is not typesafe and indicates presence of a runtime error.

The annotations are written as comments in a predefined syntax, so that they will be ignored by the compiler and the behavior of the source code will not be changed. The annotations are translated by *c2pvs* to axioms or lemmas which are used in the proofs. We

```
extern int ext_glob;
int find_z(int a,int b)
{
    int p,z;
    if(ext_glob>0)
        p=a;
    else
        p=b;
    if(ext_glob>0)
        l1: z=(a*1000)/(p-b);
    else
        l2: z=(a*4000)/(p-b);
    return z;
}
```

Figure 8.2: Example Program

```
extern int ext_glob;
int find_z(int a,int b)
{
    int p,z;
    /*pre a≥1500 AND a≤2000
    AND b ≥500 AND b ≤ 1000 end*/
    if(ext_glob>0)
        p=a;
    else
        p=b;
    if(ext_glob>0)
        l1: z=(a*1000)/(p-b);
    else
        l2: z=(a*4000)/(p-b);          ⟵
    return z;
}
```

Figure 8.3: Annotated Program

explain the method with the help of the example shown in Fig.8.2, where the value of z is computed based on the parameters a and b.

There are two parameters a and b for the given function. Let the ranges of values a and b can take at the entry to the function be,

```
1500 <= a <= 2000
500 <= b <= 1000
```

The function `find_z` can be annotated with these constraints using annotations of type *pre*-conditions denoted as *pre*. The annotated function is shown in Fig.8.3. It can be seen that there is a divide by zero error in the statement $z = (a * 4000)/(p - b);$ at $l2$ (shown by an arrow) as the expression $p - b$ evaluates to zero at this program point (Note that the statement $z = (a * 1000)/(p - b);$ at $l1$ does not cause divide by zero error). The actual steps in the proof process are further explained below.

As shown in Fig.8.1, the annotated function is given as input to *c2pvs*. The *c2pvs* translator translates the function to a state transition system encoded as PVS specification. The *pre*-condition is translated as an axiom in the PVS specification. The PVS specification generated by *c2pvs* is then loaded in PVS and typechecked. The PVS typechecker generates the following TCC to guarantee that $p - b$ is not zero at $l2$ (*sub_restr_int* is the PVS representation of the binary $-$ operator in C).

```
OBLIGATION sub_restr_int(p, b) /= 0
```

The proof of this TCC is tried using the *pre*-condition and the PVS prover commands. The proof fails, which indicates that $p-b$ is zero at $l2$ and the statement $z = (a*4000)/(p-b)$

Table 8.1: Formal annotations and their syntax

Annotation Syntax	Meaning
$/*pre$ **formula** $end*/$	Conditions true at entry to function
$/*post$ **formula** $end*/$	Conditions true at exit of function
$/*postfunc$ **formula** $end*/$	Model effect of function calls
$/*prefunc$ **formula** $end*/$	Conditions true before function calls
$/*loopinv$ **formula** $end*/$	Loop invariants

can lead to a divide by zero error. A similar TCC is generated at $l1$ for the statement $z = (a * 1000)/(p - b)$ also; but it gets proved indicating that the statement does not result into division by zero.

The annotations of type *pre*-conditions used here specify the ranges of values function parameters and global variables can take at the entry to a function. In general *pre*-conditions can be used to specify the constraints on the input data (ranges of input data) and any condition which is true at the entry to the function. It must be noted that these annotations are only used to capture the ranges of the input data and do not capture the functional specifications of each C function. Tools like PolySpace use similar *Data Range Specifications* for global variables. The different types of annotations supported by the tool and their syntax are listed in table 8.1.

Generating PVS specification of a C program involves modeling the datatypes and modeling the execution semantics. The next section describes the modeling scheme in detail.

The method can be used for runtime error detection of sequential C programs at the unit level (C function level). However, to guarantee the absence of runtime errors across the entire program, one can use compositional technique.

8.3 Modeling C Programs in PVS

8.3.1 Modeling Datatypes of C in PVS

The PVS type system has *number, real, rational, integer, and natural* as number related types [35]. The variables of these types can have values varying from -∞ to $+\infty$ (except natural where the range is 0 to $+\infty$). Unlike PVS, in C the ranges of values for datatypes are restricted. Hence we have modeled the C datatypes as restricted types (subtypes) in PVS.

Modeling Integer and Character Datatypes: Integer and character datatypes of C are modeled as subtypes of the PVS datatype *int*, restricted between the maximum and minimum representable values. For example, *int* datatype of C is modeled as a subtype *restr_int*.

```
restr_int:TYPE =
{x:int| x<=INT_MAX AND x>=INT_MIN AND INT_MAX>=INT_MIN}
```

where INT_MAX and INT_MIN are constants indicating the integer maximum and integer minimum respectively for the machine on which the C program will be executed. Similar modeling is used for other integer types and character types.

Operators on integer and character datatypes are modeled as functions with subtype arguments and PVS type return value. For example, arithmetic operators on *int* are modeled as functions with *restr_int* arguments and *int* return value. Binary $+$ on *int* is modeled as a function *add_restr_int*.

```
add_restr_int(x:restr_int,y:restr_int):int=x+y
```

Similar modeling is used for other operators on integer and character datatypes.

Modeling Floating Point Datatypes: Floating point datatypes of C (*float*, *double*, and *long double*) are modeled as subtypes of PVS datatype $real$ restricted to the normalized range [71].

For example, *float* datatype of C is modeled as a subtype *restr_float*.

```
restr_float:TYPE={x:real|((x <= MAX AND x >= MIN) OR
    (x <= -1*MIN AND x >= -1*MAX) OR (x = 0)) AND (MAX >= MIN)}
```

Here MAX and MIN are constants indicating the normalized float maximum and normalized float minimum respectively.

Operators on floating point datatypes are also modeled as functions with subtype arguments and PVS type return value. But these functions modeling the floating point operators first perform the operations with infinite precision and then the result is converted to finite precision. Hence three rounding functions *round_to_nearest_float*, *round_to_nearest_double* and *round_to_nearest_long_double* are defined, which round an infinite precision real number to the nearest representable float, double and long double number respectively. For example, binary $+$ operator on *float* is modeled as a function *add_restr_float*.

```
add_restr_float(x:restr_float, y:restr_float): real
    = round_to_nearest_float(x+y)
```

Similar modeling is used for other operators on floating point datatypes.

Generation of TCCs: The modeling scheme described above causes the typechecker to generate proper TCCs for all possible runtime errors. We will illustrate this with the help of two simple examples.

Let us consider the statement $c = a + b$; where a, b, c are *int*. The value of $a + b$ can go beyond INT_MAX or INT_MIN which can get assigned to c causing an integer overflow/underflow. Let us see how the typechecker detects this.

The statement $c = a + b$ is modeled in PVS as,

```
c = add_restr_int(a,b)
```

The typechecker finds that the type of c is *restr_int* which does not match with the type of *add_restr_int(a,b)*. Hence it generates the following TCC to ensure that the value of *add_restr_int(a,b)* does not go beyond INT_MAX or INT_MIN so that it is of type *restr_int*.

```
add_restr_int(a,b)<=INT_MAX AND add_restr_int(a,b)>=INT_MIN
```

Proving this TCC ensures that $a + b$ does not overflow/underflow in this statement.

Let us consider another statement $c = a/b$; where a, b, c are integers. There are two possible runtime errors in the statement.

1. b can be zero resulting in *divide by zero* error.
2. a/b can *overflow/underflow*.

The statement is encoded in PVS as,

```
c = div_restr_int(a,b)
```

The typechecker expects *restr_int* and *nonzero_restr_int* (*restr_int* with zero excluded) as the argument types for *div_restr_int*. But it encounters *restr_int* as the second argument of *div_restr_int*. Similarly it expects *restr_int* as the type of *div_restr_int* and encounters *int* instead of it. The typechecker generates two TCCs.

1. `b /= 0`

   This TCC is to ensure that b is not zero.
2. `div_restr_int(a,b)<=INT_MAX AND div_restr_int(a,b)>=INT_MIN`

   This TCC is to ensure that the operation a/b does not overflow/underflow.

Proving these TCCs ensures that b is not zero a/b does not overflow/underflow in this statement.

8.3.2 Modeling Execution Semantics of C in PVS

In our method, a C function is modeled as a state transition system, encoded as a PVS theory (A PVS specification is composed of theories similar to the way a C program is composed of functions). In the state transition system considered here, a state is a type consistent valuation of all the program variables and statements are the transitions between the type consistent states. A state is type consistent if the valuations of state variables are within proper ranges defined for that type. We encode the state transition system as a list of such states with each state defined in terms of the previous states. i.e. a statement S causing the $s_i \rightarrow s_j$ transition is represented by s_j where s_j is defined as a function of s_i. A program is type safe if all the reachable states are type consistent.

Modeling of State: State is modeled in PVS by a tuple of values. Each component of the tuple corresponds to a variable of the program. In general consider a C program with n variables $v_1, v_2, \ldots, v_i, \ldots, v_n$ of types $t_1, t_2, \ldots, t_i, \ldots, t_n$. The states of the program are modeled as tuples of type $[t_1, t_2, \ldots, t_i, \ldots, t_n]$. For example a state s_a is modeled as

$$s_a : state = (s_a, s_a, \ldots, s_a, \ldots, s_a)$$

Now the components of the tuple $s_a, s_a, \ldots, s_a, \ldots, s_a$ denote the values of the program variables $v_1, v_2, \ldots, v_i, \ldots, v_n$ respectively at the state s_a.

Assignment Statement: An assignment statement modifies the value of a program variable and thus changes the state of the program. It can therefore be modeled in PVS as a new state resulting from the application of a state transition on the present state.

Sequential Composition: Consider a sequence of statements $S_1;\ S_2;$ Let the current state of the program be s_a. Let the statement S_1 changes the program state to s_{a+1}. The statement S_2 now acts on the state s_{a+1}, to get s_{a+2} as the resulting state. We represent this in PVS as

$$\begin{aligned} s_{a+1} : state &= state\ transition\ for\ S_1(s_a) \\ s_{a+2} : state &= state\ transition\ for\ S_2(s_{a+1}) \end{aligned}$$

The statements after S_2 will act on the state s_{a+2} and are represented in the same manner as S_1 and S_2.

If-else Statement: Consider an if-else statement which occurs at a program state s_a as shown in the table 8.2. After the if-else statement at state s_a, the program can go to two possible states s_b (if the condition is true) or s_d (if the condition is false). The statements inside the if part will operate on the state s_b and those inside the else part will operate on s_d. Let s_c be the final state in the if part and s_e the final state in the else part. The states s_b to s_c and s_d to s_e are not reachable always and their reachability depends on the if condition. The state after the whole if-else statement s_f will be either s_c or s_e. Hence modeling the if-else statement includes modeling of all the states from s_b to s_f.

Loops: Consider the loop `while(B) S` , where B is the loop condition and S is the statement part of the loop. We take loop invariants as input from the user (A loop invariant is defined as a formula which is true before control enters the loop, remains true each time the program executes the body of the loop, and is still true when control exits the loop). Now the state after the loop can be modeled as any state satisfying the condition:

```
(loop invariant) AND NOT(loop condition)
```

Table 8.2: The C constructs and their PVS transformations

C construct	PVS transformation
$s_a : v_i = expr;$	$s_{a+1} : state = (s_a, s_a, \ldots, expr, \ldots, s_a, \ldots, s_a)$
$s_a : v_i = expr;$ $v_j = expr;$	$s_{a+1} : state = (s_a, sa, \ldots, expr, \ldots, s_a, \ldots, s_a)$ $s_{a+2} : state = (s_{a+1}, s_{a+1}, \ldots, s_{a+1}, \ldots, expr, \ldots, s_{a+1})$

$s_a : if(cond)$
{
$s_b : statements$
$s_c :$
}
else
{
$s_d : statements$
$s_e :$
}
$s_f :$
	
	$s_b : state =$ if $cond(s_a) = true$ then s_a else unreachable endif
	$s_c : state =$ state transition for last statement in if part
	(state previous to $s_c$in if part)
	
	$s_d : state =$ if $cond(s_a) = false$ then s_a else unreachable endif
	$s_e : state =$ state transition for last statement in else *part*
	(state previous to s_e in else part)
	$s_f : state =$ if $reachable(s_c)$ then s_c else s_e endif

$s_a : while(B)\ S$ $s_b :$	 *substate : type* = {*s* : *state{loop-invariant(s)}* $\wedge\neg(B(s))$}$ $s_b : substate$

The accuracy of the proofs depends on the correctness and accuracy of the user supplied invariants. Tools like STeP [20], which can generate loop invariants automatically can be used for this purpose.

The possible runtime errors inside the loop are detected by handling the statements inside the loop separately i.e. by putting the translation of these statements into another theory. This theory is named as *looptheory_i* for the i^{th} loop in the function. As the loop invariant conjuncted with loop condition is true before entry to the loop body, it is put as a *pre*-condition to these statements. Typechecking this theory generates TCCs representing possible runtime errors inside the loop, the proofs of which are tried by using the *pre*-condition. Thus a C function with n loops will generate a PVS specification with $n + 1$ theories i.e. n loop theories and one main theory.

The PVS transformations of different constructs in a C program with n variables

$$v_1, v_2, \ldots, v_i, \ldots, v_j, \ldots, v_n$$

of types

$$t_1, t_2, \ldots, t_i, \ldots, t_j, \ldots, t_n$$

respectively are illustrated in detail in the table 8.2. All the constructs are labeled with identifiers indicating the states at which they occur.

8.4 Proving TCCs in PVS and Tracing RTEs to Source Code

Most of the TCCs generated in PVS can be proved by using the axioms in the PVS theory, type predicates on the program variables and type predicates on the loop states followed by the application of prover command *grind*. The tool automates the proofs of such TCCs by generating a strategy which performs the above steps. Such a strategy is generated for each theory.

But there exists TCCs which cannot be proved this way; particularly those related to multiplication, and division. This is due to the reason that *grind* cannot handle nonlinear arithmetic. For example, let us consider that we want to prove in PVS the following

```
{-1}  a >= 0
  |-------
{1}   a * a >= 0
```

grind alone will not prove this. We need to use a lemma *le_times_le_pos* from the prelude (prelude consists of theories that are built into the PVS system), properly instantiate it and then invoke grind. The need to select and use lemmas from prelude and instantiating them makes the automatic discharge of such TCCs difficult. Such TCCs should be tried interactively.

Presence of any unproved TCC indicates the presence of possible RTEs in the original C program. Each TCC generated by PVS is identified by the state and the operation from which it is generated. The name of a TCC $s_i\_TCC_j$ indicates that it is generated from the j^{th} operation of the i^{th} state in the PVS model. As we know the number of states generated by each type of C statement, given that a TCC $s_i\_TCC_j$ is unproved we can infer the C statement causing the possible RTE from the state number i. The exact operation causing the RTE can be inferred from the operation number j. The type of RTE detected (array bound errors, divide by zero etc.) can be found out by analyzing the TCC description generated by PVS. Currently this functionality of tracing the RTEs from the unproved TCCs is not incorporated into the tool and effort is on to automate it.

8.5 Example

We now present an example to illustrate the method.

```
int average(int n)
{
     int i,sum,avg;
     /*pre n>=1 AND n<=100 end*/
     i=n;
     sum=0;
     while(i>0)
     {
      /*loopinv (i>=0) AND (i<=n) AND (n>=1) AND (n<=100) AND
      (sum = (n*(n+1)-(i+1)*i)/2) end*/
      sum=sum+i;
      i=i-1;
      }
      avg=sum/n;
      return(avg);
}
```

The function *average* finds out the average of first n natural numbers. The user supplied *pre-* condition and the loop invariant are inserted as formal annotations. The PVS specification generated for the function is shown below.

```
looptheory_1:THEORY
BEGIN
 ...
state:TYPE=[bool,void,restr_int,restr_int,restr_int,restr_int,restr_int]
s1:state
axiom_3: AXIOM  (s1`5>=0) AND (s1`5<=s1`4) AND (s1`4>=1) AND
    (s1`4<=100) AND (s1`6=(s1`4*(s1`4+1)-(s1`5+1)*s1`5)/2) AND
    great_restr_int(s1`5,0)
   ...
s3:state=IF s2`1=FALSE THEN s2 ELSE
    (s2`1,s2`2,s2`3,s2`4,sub_restr_int(s2`5,1),s2`6,s2`7) ENDIF
END looptheory_1

average:THEORY
BEGIN
.....
state:TYPE=[bool,void,restr_int,restr_int,restr_int,restr_int,restr_int]
s1:state
axiom_3: AXIOM  s1`4>=1 AND s1`4<=100
.....
s3:state=IF s2`1=FALSE THEN s2 ELSE (s2`1,s2`2,s2`3,s2`4,s2`5,0,s2`7)
    ENDIF
loopcondition_1:[state->bool]=LAMBDA(u:state): (great_restr_int(u`5,0))
loopinvariant_1:[state->bool]=(LAMBDA(u:state): ( (u`1=s3`1) AND
    (u`2=s3`2) AND (u`3 =s3`3) AND (u`4=s3`4) AND (u`7= s3`7) AND
    (u`5>=0) AND (u`5<=u`4) AND (u`4>=1) AND (u`4<=100) AND
    (u`6=(u`4*(u`4+1)-(u`5+1)*u`5)/2)))
substate_1:TYPE={s:state | loopinvariant_1(s) AND
    NOT(loopcondition_1(s))}
axiom_4: AXIOM EXISTS (x1: substate_1): TRUE
s4:substate_1
 .....
```

Table 8.3: The TCCs, their significance and proofs for function average

TCC Name	Theory Name	Significance of the TCC	How the TCC is proved
$s2\_TCC1$	looptheory_1	To ensure sum+i does not overflow/underflow inside the loop	Interactively
$s3\_TCC1$	looptheory_1	To ensure i-1 does not overflow/underflow inside the loop	$(looptheory\_1\_strategy)$
$s5\_TCC1$	average	To ensure n is nonzero in operation sum/n	$(average\_strategy)$
$s5\_TCC2$	average	To ensure sum/n does not overflow/underflow	$Interactively$

```
s6:state=IF s5‘1=FALSE THEN s5 ELSE
    (s5‘1,s5‘2,s5‘3,s5‘4,s5‘5,s5‘6,s5‘6) ENDIF
END average
```

The PVS specification consists of two theories. The first theory *looptheory_1* is the translation of the loop and the second is the translation of the function *average*. The statements inside the loop are translated to states s_1 to s_3 in *looptheory_1*. *loop invariant AND loop condition* is translated to axiom *axiom_3* in *looptheory_1*. In theory *average*, the *pre-condition* is translated to *axiom_3* and the loop invariant is translated to $[state \rightarrow bool]$ function *loopinvariant_1*. s_4 is the state after the loop defined as a constant of the type *substate_1* which is a subtype of the *state* type satisfying *loopinvariant AND NOT(loop condition)*.

The tool generates strategy *looptheory_1_strategy* for the theory *looptheory_1* and *average_strategy* for the theory *average*. The PVS specification is type checked. Table 8.3 shows the TCCs generated, their significance and how they are proved. All the TCCs generated are proved. Hence there are no *arithmetic overflows/underflows* and *divide by zero* in the function *average*.

8.6 Summary

In this chapter, we have presented an approach for detecting restricted type errors in C programs by transforming the input programs into typed PVS specifications and subsequently discharging TCCs using PVS. The method is capable of detecting errors such as array out of bound, divide by zero, overflow/underflow. In this approach, an unproved TCC has a direct correlation with the source line and it is easy to track the cause of the RTE in the source code. Also the loop invariants can be tightened to reduce the false positives.

We have also implemented a tool based on this method. The tool implemented using this technique takes MISRA [87] compliant subset of C language. Most of the software development process standards like IEC880 [65] used in developing software for safety-critical applications recommend use of programming rules which prohibit usages of unsafe constructs of a programming language. MISRA C:2004 [87] standard is a set of programming rules for C language categorized in *Required* and *Advisory* rules. These set of rules together define a subset of C Language. We have adopted this subset of C as it is a well accepted standard and one can check compliance to MISRA standard using commercial tools. Use of MISRA C also helps in meeting recommendations of standards for safe subsets of languages. Limiting the method to MISRA C subset which does not allow union datatype, pointer arithmetic and dynamic memory allocation eliminates difficulties with PVS model generation and reasoning. These features (allowed in general POSIX standard) are considered unsafe in context of safety critical software.

Chapter 9

Translation Validation of a HLL Compiler

9.1 Introduction

In the final realization of the executable code from high level language derived from the model for safety critical applications, very high levels of confidence in the correctness of executable code is essential. The two steps in the realization of object code (cf. Fig. 9.1) are:

1. Realizing a high level program from a given requirement specification and architectural specification.
2. Generating the object code using the translator (or compiler).

In Fig. 9.1, dotted lines are shown between the specification and the HLL realization to reflect several successive refinements whereas the solid arrow between the HLL and the translator shows the direct nature of the refinement (translation). To show that the object code indeed realizes the given specification, we need to establish that:

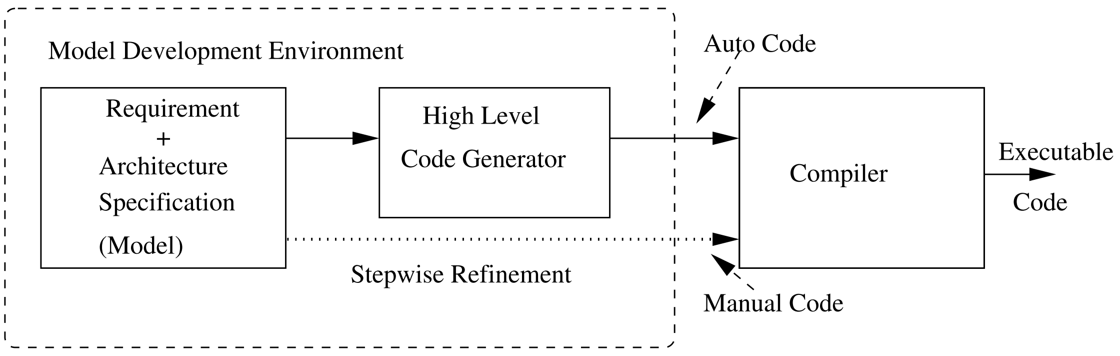

Model Development Environment
Requirement + Architecture Specification (Model)
High Level Code Generator
Auto Code
Compiler
Executable Code
Stepwise Refinement
Manual Code

Figure 9.1: Two Main Steps from Specification to Realization

1. high level program is an implementation of the given specification,
2. object code derived is a correct translation of HLL program.

In the previous chapters, we have addressed the first step. The second step depends upon the correctness of the translator. In the context of compiler development, the step corresponds to the preservation of the meaning of program by the compiler. It may be noted that even certified compilers will have several known bugs many of which could affect executable code. Thus, one has to look for solutions such as:

1. Verification of Compiler
2. Formal verification of compiled code
3. Establishing the equivalence of the source and the object code.

Compiler verification is an extremely difficult task and almost impossible (undecidable in general) and formal verification of compiled code is again extremely difficult. *Translation Validation* is an approach proposed in [98] that intrinsically realizes (3) in a pragmatic manner. Translation validation is based on *Refinement Mappings* [1] and can be used to establish that the object code produced by the compiler on a given pass, is a correct implementation of the input program. *Refinement Mappings* have been used to prove that a lower-level specification correctly implements a higher-level specification. Pnueli et al. [98] proposed the technique of *Translation Validation* to show that a program in SIGNAL [9] – a synchronous language – is correctly translated into C – an asynchronous language.

In this chapter, we demonstrate a methodology supported with tools to establish *equivalence* of the given source program with the corresponding object code. The rationale is based on the fact that usually in safety-critical embedded applications, it would suffice to establish correctness of compilation of a finite set of programs. In our approach, we find out whether the object code generated can be proved to be an implementation of the source program (restricted to an industry standard subset). For this, both the source and the object programs are brought under a common semantic framework of FTS and then it is shown that the transition system of the object program is a refinement of the transition system of the source program. The contribution of the chapter lies in the development of a methodology for Object Code Validation (OCV) with tool support for validating translations from a general purpose HLL to assembly language. We have demonstrated the use of this methodology in translation validation of a compiler in [14].

Rest of the chapter is organized as follows: Section 9.2 gives an overview of translation validation and the related work. In section 9.3, an outline of the proposed OCV method is given. This is followed by a detailed treatment of refinements, proof obligations in STeP and a simple illustrative example of the method in section 9.4. In section 9.5, we give an example wherein the method detects translational errors. In section 9.6 implementation issues regarding support tools of OCV are discussed.

9.2 Translation Validation: An Overview

For establishing the correctness of a compiler, one has to prove that the compiler always produces target code that correctly implements the source code. Owing to the intrinsic complexities of compiler verification, an alternative referred to as *Translation validation* has been explored in [98]. In this approach, each individual translation (i.e. a run of the compiler) is followed by a validation phase which verifies that the target code produced on this run correctly implements the source program. Such a possibility is particularly relevant for embedded systems where there is a need to execute a finite set of target programs. It must be pointed out that the validation task becomes increasingly difficult with the increase of sophistication and optimizations methods like scheduling of instructions as in RISC architectures or methods of code generation/optimization for super-scalar machines. In [98], the authors demonstrated the the practicability of translation validation for a translator/compiler that translates the synchronous language Signal to C without any optimizations[1]. The method exploits the special features of the SIGNAL compiler:

1. Each program consists of an initialization followed by an infinite loop consisting of phases like, calculating clock expressions,reading inputs, computing outputs, writing outputs and updating *previous* expressions.

2. The compiler translates a program structurally.

The question that arises is:

> *Is it possible to apply the above technique to a non-synchronous language (HLL) that does not use such structural translation?*

It is very clear that by the very facts of difficulties of compiler verification mentioned already, we cannot extend the method in an unconstrained manner. The basic characteristics of the underlying language and its translator that we have exploited are:

1. Source program satisfies *safeness* constraints (Language Subsets).

2. The programs also pass through some of the well-established metrics of software engineering required for certification.

3. The compiler is a certified industrial compiler that does not use complex optimizations; the assembler is a simple translator having almost a one-one correspondence between assembly and machine code.

The above characteristics are indeed the minimum requirements imposed on translators used in critical embedded applications.

[1]In [97], extension of the approach for TNI SIGNAL compiler is explored.

9.2.1 Fair Transition System

The common semantic framework viz. FTS [81], is formally described as $\mathcal{F} = (V, \Theta, \Gamma, E)$ where,

- $V = \{u_1, u_2 \cdots u_n\} \subseteq \nu$: A finite set of system variables consisting of *data variables*, and *control variables* and ν is the *vocabulary*.
- Θ: The initial condition characterizing the initial states.
- Γ: A finite set of transitions. $\tau \in \Gamma$ is a function $\tau : \Sigma \mapsto 2^{\Sigma}$ mapping each state $s \in \Sigma$ into a set of states $\tau(s) \subseteq \Sigma$.
- $E \subseteq V$: A set of externally observable variables.

A computation in FTS, $\mathcal{F}$, denotes an infinite sequence $\sigma =< s_0, s_1, s_2 \ldots >$, where $s_i \in \Sigma$ for each $i \in \mathcal{N}$ iff $s_0 \models \Theta$ and $(s_i, s_{i+1}) \models \Gamma$.

9.3 An Overview of the OCV Method

Our method for OCV consists of:

1. Translating the given HLL program into SPL(*Simple Programming Language*) used in STeP or FTS.
2. Translating the compiler generated assembly language program corresponding to the above HLL program to SPL or FTS.
3. Deriving the *interface mapping* using the symbol table generated by the compiler.
4. Using STeP to show that the FTS for the object code is a refinement of the FTS for the source program.

The overall OCV scheme is diagrammatically shown in Figure 9.2. The feasibility of our approach relies on the following aspects:

1. Restricting the language subset to *safe subsets* enforces good structural relations on the FTS for the source and the object program since we also assume that there are no complex optimizing transformations.
2. The above structure also provides support for the *Interface Mapping* of variables. For instance, translations in an actual compiler like GNU gcc, that has options (eg, -gstabs, -gstabs+ *etc.*) for producing debugging information can be effectively used to derive correspondences between variables in the source and the object program.
3. Use of interactive theorem provers such as STeP [20] or PVS [35] to establish the equivalence.

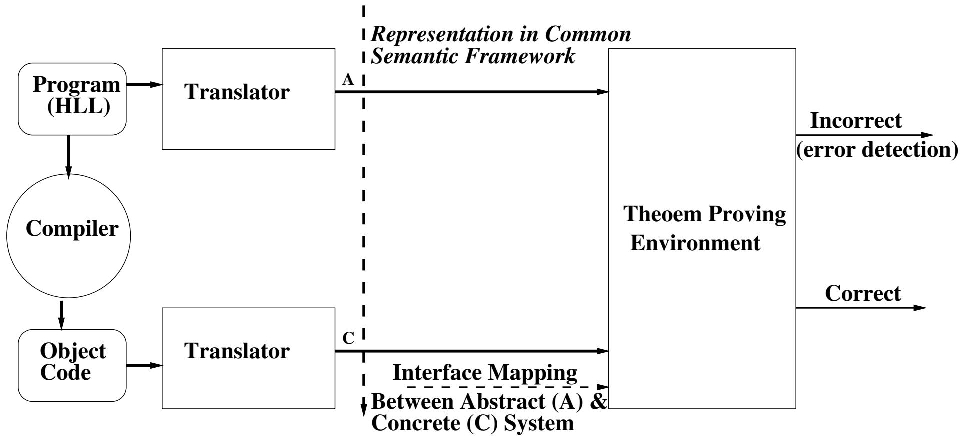

Representation in Common Semantic Framework
Program (HLL)
Translator
A
Incorrect (error detection)
Compiler
Theoem Proving Environment
Correct
Object Code
Translator
C
Interface Mapping Between Abstract (A) & Concrete (C) System

Figure 9.2: Overall Object Code Verification Scheme

9.4 Formal Description of the OCV Method

Firstly, the HLL program and it's object code are translated into FTS using the two translators shown in Figure 9.2. The two fair transition systems obtained become the input to the theorem proving tool STeP to carry out the proof of correctness of translation. The other input required is the *Interface Mapping* (cf. in Fig.9.2) that provides a mapping between the variables in the HLL program and the object code. Although FTS is the final representation used in the proof, it is also possible to first translate the HLL and it's object code into SPL, which is accepted by STeP as input. The translators shown in Figure 9.2 are designed to take one HLL program and it's object code as input and produce the corresponding SPL programs. The representations of the HLL and it's object code, in the form of FTS, could then be obtained using STeP. However, as explained later, we have found that in the case of object code it is easier to directly translate it to FTS. We illustrate the above concepts using a simple HLL program given below:

```
int test(int a)
    { int i,j=1,k=2;
    if(a)
        i=j*i +k;
    else
        i=k*i +j;
}
```

The FTS of the above HLL program is shown in Fig.9.3 and is referred to as *Abstract System* denoted by A=$(V_A, \Theta_A, \Gamma_A, E_A)$, where $V_A = \{pi0, i, j, k, l, n, a\}$, $\Theta_A = \{pi0 = 0\}$, $\Gamma_A = \{T_1, T_2, T_3, T_4, T_5\}$ and $E_A = \{a, l, i, k, j\}$; pi0 denotes program location counter. The transition system shown in Fig.9.4 is obtained from the object code of the above HLL program. For lack of space, the object code is not shown here. In the sequel, the FTS corresponding to the object code is always referred to as *Concrete System* and is denoted by *C*=$(V_C, \Theta_C, \Gamma_C, E_C)$

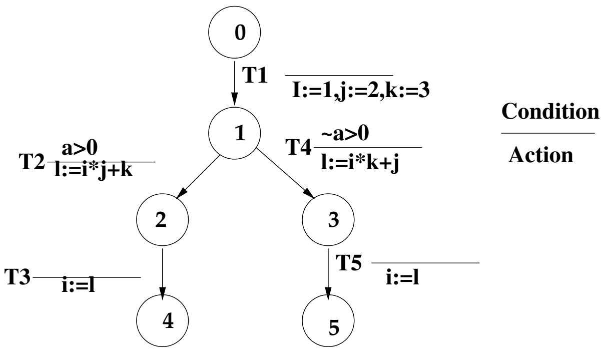


Figure 9.3: Transition System for the Abstract System

where $V_C = \{$ pi0, ebp_4, ebp_8,ebp_12,ebp_16, ebp8,ecx,edx,eax,esp $\}$,$\Theta_C = \{pi0 = 0\}$, $\Gamma_C = \{T_1, T_2, \ldots, T_{15}\}$ and $E_C = \{ebp\_4, ebp\_8, ebp\_12, ebp\_16, ebp8\}$. The *Concrete System* variables ebp_4, ebp_8, ebp_12, ebp_16 and $ebp8$ correspond to the variables in the *Abstract System*. The correspondence between variables of the *A* and *C* is $\{i \mapsto ebp\_4, j \mapsto ebp\_8, k \mapsto ebp\_12, l \mapsto ebp\_16, a \mapsto ebp8\}$. Thus, $V_A \subset V_C$. Even though in this example there is a direct one to one correspondence between the observables of the two systems, in general it need not be so. For example, HLL may support an abstract data type like a long integer (64 bit), which when mapped to a low level assembly program may be represented as two 32 bit integers (d1,d2) and the mapping will be an expression ($2^{32} * d2 + d1$). It can be also observed that the number of states in the two transition systems are different ($\sharp\Sigma_C \geq \sharp\Sigma_A$). A state in the HLL program (or it's FTS) is said to correspond to a state in the object program (as represented by its FTS) when they agree on the values of the observable variables. Formally, if $s_A \in \Sigma_A$ and $s_C \in \Sigma_C$ are two states of HLL and object program then $s_A \mapsto s_C$ iff $\forall v_A \in E_A \bullet v_A(s_A) = v_C(s_C)$ where $v_C \in E_C \wedge v_A \mapsto v_C(s_C)$. Here $v(s)$ means the value of variable v interpreted in state s. Thus, in Figure 9.4 the shaded nodes *{0,7,8,14,15}* are the states which correspond to states *{0,2,4,3,5 }* in abstract system in that order. The states of the *Concrete System* that do not have a correspondence with those of the *Abstract System* are unobservable and hence are local to it.

9.4.1 Correctness of Translation

Let us consider an *Abstract System A* representing a HLL program and *Concrete System C* representing the corresponding object code. The system *A* can be viewed as a specification for the implementation *C*. A refinement mapping [1] is defined as a function ($f : \Sigma_C \mapsto \Sigma_A$) which maps concrete system states to abstract system states. If the translation is correct, *C* will preserve the essential features of *A* except for:

- The concrete system need not agree with the abstract system on the values of all variables. The refinement relation singles out the *observable variables* whose behavior should be preserved.

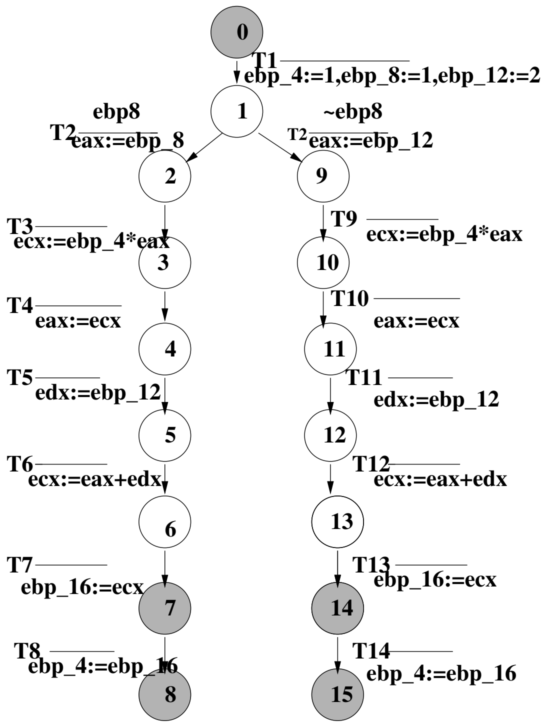


Figure 9.4: Transition System for the Concrete System

- In the course of computation, the concrete system may require data movement in temporary locations (registers). This leads to the possibility of loosing one to one correspondence between states in the two systems.
- The abstract system can operate in terms of high level abstract data types, while the concrete version is restricted to only those data types available in the particular architecture. Consequently, one should not always expect one to one correspondence between the concrete observable variables and the abstract ones.

From the theory of refinement mapping [98, 1], we have:

- if $f : \Sigma_C \mapsto \Sigma_A$ is an inductive refinement mapping from C to A then C is said to be a refinement of A i.e $C \sqsubseteq A$
- A refinement mapping $f : \Sigma_C \mapsto \Sigma_A$ is called inductive iff **B-INV** rule given below is satisfied.:

  R1. *Initiation:* $s \models \Theta_C \rightarrow f(s) \models \Theta_A$, $\forall\ s \in \Sigma_C$ and

  R2. *Propagation:* $(s, s') \models \Gamma_C \rightarrow (f(s), f(s')) \models \Gamma_A$ for all $s, s' \in \Sigma_C$

Given two FTSs, *A* and *C*, we have to show that *C* is a correct implementation of *A* or in other words *C* refines *A*. We assume that $E_A = V_A - \{pi0\}$. Let $\alpha : V_A \rightarrow \varepsilon(V_C)$ be a substitution that replaces each abstract variable $v \in V_A$ by an expression ε_v over the concrete variables. Such a substitution α induces a mapping $\tilde{\alpha}$ between states. To show that *C* refines *A* it is required to show that [1] R1: $\Theta_C \rightarrow \phi$, R2: $\forall \tau \in \Gamma_C \bullet \{\phi\}\tau\{\phi\}$, R3: $\Theta_C \rightarrow \Theta_A[\alpha]$ and R4: $\phi \rightarrow \phi[\alpha]$, where ϕ be an assertion on C and $\alpha : V_A \rightarrow \varepsilon(V_C)$ is a substitution. The rules R1-R2 express the requirement that ϕ is an invariant of system C and R3-R4 express the requirement that $\tilde{\alpha}$ is an inductive refinement mapping. We define ϕ to be an invariant defining the conditions under which observable variables are changed by the computation.

9.4.2 Proof of Validation using STeP

Let u_A be an observable variable in *A* and u_C be the corresponding variable in *C*. Consider a state s_i in the abstract system where u_A is defined. Let $l_{u_A}(s_0 \hookrightarrow s_i)$ be the path condition (conjunction of all predicates on the path $(s_0 \hookrightarrow s_i)$ and $cond(u_A)$ be the disjunction of all such path conditions, because there can be more than one path from s_0 to s_i. Then an invariant can be defined by $\phi : at\_s_i \rightarrow cond(u_A) \wedge u_A = \ldots$. Thus taking the FTS in Fig. 9.3, we can have $\phi : pi0 = 2 \rightarrow a > 0 \wedge l = i*j+k$ and $\phi : pi0 = 3 \rightarrow \neg(a > 0) \wedge l = i*k+l$. Such invariants can also be defined for the concrete system. The fact that these invariants are indeed true in the respective systems can be verified by the B-INV rule. If for any transition $\tau \in \Gamma$, $\{\phi\}\tau\{\phi\}$ is not established, one can generate a weakest precondition (WPC)[37] which should hold good before the transition is taken so that ϕ remains true after the transition. The WPC itself can be checked by applying B-INV. These can be applied repeatedly till it is proved or disproved.

We define the substitution function $\alpha : V_A \rightarrow \varepsilon(V_C)$. This function defines the mapping between an abstract variable to its counterpart in the concrete system. In STeP this mapping is expressed by Simplify or Rewrite rules. These simplification rules are declared with the **SIMPLIFY, REWRITE** keywords. The **SIMPLIFY** rules are automatically and exhaustively applied when STeP simplifier is invoked. **REWRITE** rules are applied interactively. We also define a mapping of states in the two system where the values of the observables in the two systems are same. Now if $\phi(u_i^C)$ is an invariant in the concrete system for a concrete variable u_i^C and $\phi(u_i^A)$ is an invariant in the abstract system then if the translation is correct $\phi(u_i^C) \rightarrow \phi(\alpha(u_i^A))$ must be true by rule R4. This should be true for all observable variables. The technique by which this is carried out is the MON-I[81] rule (modus ponens) which says if p is true and $p \rightarrow q$ then q is true.

For the system shown in Fig. 9.3, we prove the following invariants (PROPERTY PA1-PA4) by using B-INV and WPC rules. For the systems shown in Fig. 9.3 and Fig 9.4 the script showing the proof requirement for correct implementation is given below:

```
(*These are the invariants of the Abstract System and
are shown in the syntax accepted by STeP.*)
PROPERTY PA1: [](pi0 = 2 --> a > 0 /\ l = i*j +k)
PROPERTY PA2: [](pi0 = 4 --> a > 0 /\ i = 1)
PROPERTY PA3: [](pi0 = 3 --> ~(a > 0) /\ l = i*k +j)
PROPERTY PA4: [](pi0 = 5 --> ~(a > 0) /\ i = 1)
( * Properties PC1-PC4 are invariants of the Concrete System *)
PROPERTY PC1:[](pi0=7 --> ebp8 >0 /\ (ebp_16=ebp_8*ebp_4 + ebp_12))
PROPERTY PC2:[](pi0=8 --> ebp8 >0 /\ ebp_4 = ebp_16)
PROPERTY PC3:[](pi0=14 --> ~(ebp8 > 0) /\ (ebp_16=ebp_12*ebp_4 +ebp_8))
PROPERTY PC4:[](pi0=15 --> ~(ebp8 > 0) /\ ebp_4=ebp_16)
(*Interface  mapping between abstract variables i,j,k,l,a and the
  concrete variables -4(ebp), -8(ebp), -12(ebp),-16(ebp) and 8(ebp).
  Here -1 means the variable is unobservable *)
value i:int*int --> int
value j:int*int --> int
value k:int*int --> int
value l:int*int --> int
value a:int*int --> int
SIMPLIFY S1: i(pi0, ebp_4) ---> if pi0 >= 0 then ebp_4 else -1
SIMPLIFY S2: j(pi0, ebp_8) ---> if pi0 >= 0 then ebp_8 else -1
SIMPLIFY S3: k(pi0, ebp_12) ---> if pi0 >= 0 then ebp_12 else -1
SIMPLIFY S4: l(pi0, ebp_16) ---> if (pi0=7 \/ pi0=8 \/ pi0=14
             \/ pi0=15) then ebp_16 else -1
SIMPLIFY S5: a(pi0, ebp8) ---> if pi0 >= 0 then ebp8 else -1
(* Axioms A1-A4 axiomatize the correspondence between observable states *)
control pca:[0..5] (* control variable *)
AXIOM A1: pi0=7 <==> pca=2
AXIOM A2: pi0=14 <==> pca=3
AXIOM A3: pi0=8 <==> pca=4
AXIOM A4: pi0=15 <==> pca=5
(* Properties P1-P4 are invariants of the Abstract System (written with
   mapping). Proof Obligation for correct refinement *)

PROPERTY P1:[](pca = 2 -->a(pi0,ebp8) > 0 /\ l(pi0,ebp_16) =
              j(pi0,ebp_8)*i(pi0,ebp_4) + k(pi0,ebp_12))
PROPERTY P2:[](pca = 4 -->a(pi0,ebp8) > 0 /\ i(pi0,ebp_4) =
              l(pi0,ebp_16))
PROPERTY P3:[](pca = 3 --> ~(a(pi0,ebp8) > 0) /\ l(pi0,ebp_16) =
              k(pi0,ebp_12)*i(pi0,ebp_4) + j(pi0,ebp_8))
PROPERTY P4:[](pca = 5 --> ~(a(pi0,ebp8) > 0) /\ i(pi0,ebp_4) =
              l(pi0,ebp_16))
```

Here properties PC1-PC4 are invariants of the concrete system. These are proved using B-INV and WPC as was done in proving properties PA1-PA4 for the abstract system. Properties P1-P4 are the properties of the abstract system (PA1-PA4) but written with substitution function. These properties are again proved using the properties PC1-PC4 and MON-I rule; note that premise R3 can be trivially proved.

9.5 Illustrative Example with Translation Error

The above technique was tested on experimental basis to a number of test programs written in C language. The translations of C programs were deliberately seeded with errors to test the efficacy of the method. All seeded errors could be detected by carrying out the required proofs. This was detected during the proof.

SPL Representation of HLL Source Program (Abstract System)

```
macro LAST:int where LAST=1024
(*
macro C1: [0..999] where C1=10
macro C2: [0..999] where C2=15
macro C3: [0..999] where C3=20
macro const:int where const=1
*)
in param:int
local v1,pav1:[-LAST..LAST]
local v2,v3:[0..999]
local C1: [0..999] where C1=10
local C2: [0..999] where C2=15
local C3: [0..999] where C3=20
local const:int where const=1
case4:: [
v1:=param;
v2:=v2*8;

l0: if v2 >= 1 /\ v2 <= 99 then
[ v3:=C1;
l1: skip ]
    else
    [ if v2>= 100 /\ v2 <= 199 \/ v2=201 then
     [ v3:=C2+C1;
l2:        skip ]
      else
      [ if v2=0 then
        [ v3:= const;
l3:       skip ]
        else
        [ if v2=200 then
          [ v3:=v2 div 4;
l4:         skip]
          else
          [ if v2=202 then
            [ v1:=pav1;
l5:           skip]
            else
            [ if v2>=203 /\ v2 <= 999 then
              [ v3:=v2+ const;
l6:             skip]]]]]];
    pav1:=v1;
l7: skip]
```

In the abstract system

$$
\begin{aligned}
&A = \{V, \Theta, \Gamma, E\} \text{ where}\\
&V = \{pi0, param, v1, v2, v3, C1, C2, C3, pav1\}\\
&\Theta = pi0 = 0\\
&\Gamma = \text{the set of transitions and}\\
&E = \{v1, v2, v3, C1, C2, C3\}
\end{aligned}
$$

The following are the invariants of the abstract system as included in the specification file (SPEC file) for the STeP.

```
PROPERTY P1: l1==>1<=v2 /\ v2 <= 99 /\ v3=C1
PROPERTY P2: l2==>((100<=v2 /\ v2<= 199) \/ v2=201) /\ v3=(C2+C1)
PROPERTY P3: l3==>v2=0 /\v3=const
PROPERTY P4: l4==>v2=200 /\ v3 = (v2 div 4)
PROPERTY P5: l5==>v2=202 /\ v1=pav1
PROPERTY P6: l6==>203<= v2 /\ v2 <= 999 /\ v3=(v2+const)
```

These invariants are proved using the rule `repeat(B-INV;Simplify;Undo;WPC)`.

SPL Representation of Object Program(Concrete System)

```
in g0:int
local r1,r2,r3,r4,r5,r6,r7,g1,g2,g3,g4,g5,g6,g7,v1,v2,v3,pav1:int
local temp:int
prog::
[ r5:=g0;temp:=r4 * 8;r4 := temp;r3:=r4;
l1:if r4>0 then
   [ g3:=99;
l2:  if r3 > 99 then
     [ g1 := 199;
l3:    if r3 > g1 then
```

```
       [ g6:=200;
         if r3 != g6 then
         [ g5 := 201;
l4:        skip;
l5:        if r3 = g5 then
           [ g6:=25;v3:=g6;
l6:          skip]
           else
           [ g3:=202;
l7:          if r3 != g3 then
             [ g6:= 203;
l8:            if g6 >= r3 then
               [g5:=v1;g4 := r4+g5;v3:=g4;
l9:             skip]
               else
               [ pav1:=r5;
l10:             skip]]
             else
             [ g1:= pav1;v1:=g1;
l11:           skip]]]
          else
          [ g6:=r4 div 4;v3:=g6;
l12:        skip]]
        else
        [ g7:=100;
l13:      if g7 <= r3 then
          [g6 := 25;v3 := g6;
l14:       skip]]]
     else
     [ if 1 <= r3 then
       [ g7:=10;v3:=g7;
l15:     skip]]]
    else
    [g4:=12;v3:=g4;
l16: skip]]
```

In the above concrete system,

$$
\begin{aligned}
&C = \{V, \Theta, \Gamma, E\} \\
\text{where}& \\
&V = \{pi0, r3, r4, v1, C1, C2, C3, pav1\} \\
&\Theta = \{pi0 = 0\} \\
&\Gamma = \text{the set of transitions} \\
&E = \{r3, v1, v2, v3, C1, C2, C3, pav1\}
\end{aligned}
$$

The following are the invariants of the system as included in the specification file (SPEC file) for the STeP. These are also proved using the same rules as in case of abstract system invariants.

```
macro C1: int where C1=10
macro C2: int where C2=15
macro C3: int where C3=20
macro const:int where const=12
control pca:[0..8]
AXIOM A1: []Forall i:int.(i=r4-->i=r3)
PROPERTY P1: l15==>1<=r3 /\ r3 <= 99 /\ v3=10
PROPERTY P2: l14==>(100<=r3 /\ r3<= 199)    /\ v3= 25
PROPERTY P3: l6==>r3=201 /\ v3=25
PROPERTY P4: l16==>r3<=0 /\ v3=12
PROPERTY P5: l12==>r3=200/\v3=(r4 div 4)
PROPERTY P6: l11==>r3=202 /\v1=pav1
PROPERTY P7: l9==>203<=r3 /\v3=(r4+v1)
```

The refinement mapping between abstract system variables and the concrete system variables and the proof obligations are shown below.

```
local V1,V2,V3,PAV1:int
(* Interface Mapping between Abstract System variables
and Concrete System variables *)
value fv1:int*int*int-->int
value fv2:int*int*int-->int
value fV2:int*int*int-->int
value fv3:int*int*int-->int
value fpav1:int*int*int-->int
SIMPLIFY S1: fv1(pi0,V1,v1) ---> if pi0 >= 0 then v1 else -1
SIMPLIFY S2: fv2(pi0,V2,r3) ---> if pi0 >= 0 then r3 else -1
SIMPLIFY S3: fv3(pi0,V3,v3) ---> if pi0 >= 0 then v3 else -1
SIMPLIFY S4: fpav1(pi0,PAV1,pav1) ---> if pi0 > 0 then pav1 else -1
```

```
(* State Correspondence between Abstract States and Concrete States *)
AXIOM M1:[](pca=1<-->l15)
AXIOM M2:[](pca=2<-->l14)
AXIOM M3:[](pca=3<-->l6)
AXIOM M4:[](pca=4<-->l16)
AXIOM M5:[](pca=5<-->l12)
AXIOM M6:[](pca=6<-->l11)
AXIOM M7:[](pca=7<-->l9)
(* Proof Obligation for correct translation *)
PROPERTY P1: pca=1==> 1 <= fv2(pi0,V2,r3)  /\ fv2(pi0,V2,r3)<= 99 /\
             fv3(pi0,V3,v3)=10
PROPERTY P2: pca=2==>(100<=fv2(pi0,V2,r3) /\ fv2(pi0,V2,r3)<= 199) /\
             fv3(pi0,V3,v3)=(C2+C1)
PROPERTY P7: pca=3==>fv2(pi0,V2,r3)=201 /\ fv3(pi0,V3,v3)= (C2+C1)
PROPERTY P3: pca=4==>fv2(pi0,V2,r3)=0 /\fv3(pi0,V3,v3)=const
PROPERTY P4: pca=12==>fv2(pi0,V2,r3)=200/\ fv3(pi0,V3,v3)=
             (fv2(pi0,V2,r3) div 4)
PROPERTY P5: pca=6==>fv2(pi0,V2,r3)=202 /\ fv1(pi0,V1,v1)=
             fpav1(pi0,PAV1,pav1)
PROPERTY P6: pca=7==>203<= fv2(pi0,V2,r3) /\ fv2(pi0,V2,r3) <= 999 /\
             fv3(pi0,V3,v3)=(fv2(pi0,V2,r3)+ fv1(pi0,V1,v1))
```

It is found that the property P3 and P6 could not be verified. The failure of P3 is because in the abstract system we have a state v3=const, with the path condition $l3 \longrightarrow v2 = 0 \wedge v3 = const$. Whereas in the concrete system, for the corresponding state where v3=const the path condition is $v2 \leq 0$. The failure of P6 is because the upper bound of v2 is 999 which is missing in the concrete system.

9.6 System for OCV: Implementation Features

The main tasks of the implementation lie in generating the FTS for the source and object code, extracting interface mapping, and the algorithm for proof. Extraction of interface mapping information is done closely to that discussed in Section 9.3 so as to achieve the mapping semi-automatically. The algorithm for proof of validity follows on the lines of STeP theorem prover [20]. The translator for HLL produces SPL output while that for object code produces Fair Transition System (FTS) for reasons explained in section 9.6.2. The features of generating SPL/FTS and the underlying modeling are discussed in the following.

Table 9.1: HLL-SPL mapping

Annotated HLL Declaration	**SPL Translation**
Multiply (X,Y: in Matrix,Z:out Matrix) –# derives Z from X,Y	in X,Y:Matrix out Z:Matrix Z:=Multiply_Z(X,Y)

Exchange(X,Y: in out float)
–# derives X from Y &
–# Y from X;
	in X,Y:rat
	out_mX,_mY:rat
	_mX:=Exchange_X(Y)
	X:=_mX
	_mY:=Exchange_Y(X)
	Y:= _mY

9.6.1 Generating SPL/FTS for the HLL Source

The methodology was used in translation validation of a specific compiler. Some of the features of the translation scheme are discussed below:

- Each function/procedure is translated into a SPL procedure.
- Most of the HLL has support for many types of data structures like records, aggregates, enumerated types which do not have corresponding types in SPL. Hence, the translators implemented for HLL handle only the basic data types which have corresponding types in SPL.
- The function invocations in HLL and object code are translated to corresponding function invocations in SPL or FTS. These functions are required not to have any side-effects.
- Procedures in languages like Ada are modeled as multiple functional assignments to **out** variables.

The **in** and **out** variables and their mutual dependencies are required to be explicitly annotated in the programs input to the translators, so that the functional assignment relationships between the **in** and **out** variables can be inferred. Some illustrative examples are shown in Table 9.1. Let us consider a procedure to multiply two matrices and the result returned in a third matrix. In SPL, the data type *Matrix* is to be specified as user defined type with corresponding axioms (not shown). A typical annotation for Ada declaration and the corresponding SPL code is shown in Table 9.1.

Consider a procedure *Exchange* for exchanging two variables as shown in Table 9.1. which is called from a procedure being analyzed. The convention of naming the functions in SPL modeling the procedure is <procedure name_<variable name which is exported > (list of variables modifying the exported variable).

Table 9.2: Assembly Instructions to FTS mapping

Assembly Instruction	**SPL/FTS Statement,**	**Declaration & Rule**
cmpi src1,src2	cc:=cmp(src1,src2)	value $cmp : int * int \longrightarrow int$ SIMPLIFY $cmp(src1, src2) \rightarrow$ if($src1 < src2$) then 4 else if($src1 = src2$) then 2 else if ($src1 > src2$) then 1 else 0
shli len,src,dst	dst:=src*power(2,len)	value $power : int * int \longrightarrow int$
shri len,src,dst	dst:=src div power(2,len)	REWRITE $\forall m, n : int \bullet$ power$(m, n) \rightarrow$ if($n = 0$) then 1 else if($n = 1$) then m else $m \star power(m, n-1)$

9.6.2 Generating FTS for the Object Code

The object code is translated directly into FTS for reasons explained below. We have used the instruction set of i960 processor for illustration. The iterative type of statements from HLL program may be translated to object code by using conditional and unconditional branching statements because i960 processor has only binary branch statements and no *loop* statements. Since the predicates in the *loop* type of statement in the HLL may be complex i.e. containing conjunction and disjunction of variables, it becomes very difficult to reconstruct a loop type construct in SPL from the form in which exists it in object code. This *reverse engineering* can only be done through extensive flow graph analysis. However, since the Fair Transition System syntax supports *goto*, it is straight forward to translate such constructs into Fair Transition Systems rather than into SPL. Hence the translator implemented for object code produces FTS directly instead of an SPL program.

In the implementation of the translator from assembly instruction to FTS the main task lies in modelling the assembly instructions and some illustrative instructions modelling are discussed in the following. We have considered i960 (RISC) processor for the purpose of illustration. Since many of the instructions in the instruction set have implicit operation, it is required to use SIMPLIFY/REWRITE rules to model the effect of the instruction. Some of the illustrative instructions are modelled as shown in Table.9.2. Let us take for example the `cmpi src1, src2` instruction which compares two integers and sets the condition flag `cc` to 4, 2 or 1 depending on the condition `src1 < src2`, `src1=src2` or `src1 > src2`. This is modelled as an assignment
`cc:= cmp(src1,src2)`.

Let us consider the example of arithmetic shift `shli` and `shri` shown in Table 9.2. A multiplication or division by some power of two is usually implemented by an arithmetic shift left or right respectively when translated by the compiler.

The i960 processor supports movement of data between floating registers fp0-fp3 and other registers like r0-r15 and g0-g15. Since in our implementation the register sets r0-15 and g0-g15 are declared as integer types, operations involving this requires AXIOMs : $\forall m : rat \bullet (Real(Int(m)) = m)$ and $\forall m : int \bullet (Int(Real(m)) = m)$.

9.7 Summary

The object code validation is a task requiring special skills even in the presence of mechanized theorem provers. The implementation of the translators to SPL/FTS is a major step in reducing the total effort involved in object code validation. Human interaction is still required in constructing the interface mappings and in carrying out the proofs.

The technique generally works fine if there is a structural correspondence between the flow graph of the two programs. The common semantic representation should have capability to handle different type of data structures normally used in HLLs like C or Ada. Each verifiable unit, which is a function or procedure in our case, should be small enough, so that it is easily possible to establish state correspondence and construct interface mapping. This is not a problem if the software is nicely modularized following good software engineering practices where each module has a small cyclomatic number. This is generally a requirement for software to be used in safety-critical system. It is also seen that auto code generators follow systematic patterns for code generations.

The process of validation requires skills with theorem provers. However, even with our preliminary experience we find it to be very useful for the validator. On the fly validation [88] aids in generating correct code as large fraction of of target-dependent errors in compilers can be detected. Program annotation and assertions aid in the proof.

Chapter 10

Conclusions, Future Work and Main Contribution

10.1 Conclusion

The process of developing a safety-critical system which is tractable and verifiable, must be based on systematic techniques. The traditional life cycle model can be described by the following three layers:

- Layer 1: Requirements Specification: In this layer, user requirements and architectural constraints are captured preferably in a formal specification language. This is an important layer for capturing the relevant functionality and its adequate formalization in precise mathematical definitions to reduce ambiguity. This layer serves the basis of abstracting a model by defining the context of the system. The issues which are of interest here are ensuring completeness and consistency of the requirements.
- Layer 2: System Model & Design: Adequate specification resulting from the first phase forms the basis for this phase which comprises the derivation of an abstract implementation. This phase usually follows the notion of stepwise refinement. It will include refinement of the operational behavior, and verification to ensure correct development. This layer thus demands extensive support for modeling, simulation, testing, code generation/synthesis and verification.
- Layer 3: System Model to Executable Model : In this layer the actual implementation is carried out using a programming language and a compiler. The outputs of this phase are executable codes. Considerable effort goes into generating optimized code for reconfigurable architectures. Issues concerning this layer would be to validate that the executable code corresponds to design.

For a correct realization the above process demands the process of rigorous verification at each of these layers. In this thesis, we have addressed some of the concerns for the *vertical* *transformations* through each of these layers. We have shown a formalization of Statecharts which is a visual modeling notation for used specifying reactive systems by providing a translation scheme to the imperative reactive language ESTEREL . We have actually built a tool titled *Programming Environment for Reactive Systems* which has an editor, simulator and translator to support modeling, verification and code generation via ESTEREL framework for supporting model based system design.We have also extended the Activity Diagram to support compensations in Workflow systems. We have provided a simulation semantics for Activity Diagrams in terms of equivalent Esterel constructs which can be used to provide a framework to automatically generate code from Activity Diagrams.

We have shown a language framework called *ScriptOrc* to model choreography in distributed service oriented architecture. We have shown how *abstract scripts* can be used to show service conversations. The idea of roles and the critical set of roles have been identified in definition of the scripts. We have provided a reactive framework for the language which captures the notion of abortion of scripts due to events and timeouts. We have also provided an operational semantics of the language.

We have presented an approach for detecting restricted type errors in C programs by transforming the input programs into typed PVS specifications and subsequently discharging TCCs using PVS. The method is capable of detecting errors such as array out of bound, divide by zero, overflow/underflow. In the proposed approach, an unproved TCC has a direct correlation with the source line and it is easy to track the cause of the RTE in the source code. Also the loop invariants can be tightened to reduce the false positives.

We have presented a technique for object code validation using the principle of Translation Validation. The object code validation is a task requiring special skills even in the presence of mechanized theorem provers. The implementation of the translators to SPL/FTS is a major step in reducing the total effort involved in object code validation. Human interaction is still required in constructing the interface mappings and in carrying out the proofs.

The technique generally works fine if there is a structural correspondence between the flow graph of the two programs. The common semantic representation should have capability to handle different type of data structures normally used in HLLs like C or Ada. Each verifiable unit, which is a function or procedure in our case, should be small enough, so that it is easily possible to establish state correspondence and construct interface mapping. This is not a problem if the software is nicely modularized following good software engineering practices where each module has a small cyclomatic number. This is generally a requirement for software to be used in safety-critical system. It is also seen that auto code generators follow systematic patterns for code generations.

10.2 Limitations and Future Work

Model driven design frameworks which support requirement specification, modeling, simulation, test coverage, formal verification and automatic code generation are becoming available to the industrial users. In this thesis, we have addressed few important issues regarding formalization and realization in vertical transformation of models to executable code. We have not addressed the aspects of code generation from synchronous languages to traditional high level languages. One of the reasons is that generation of C code from synchronous languages is well understood. However the distribution of generated code in a distributed architecture needs attention. This is a challenge that needs to be addressed.

We have discussed the reactive requirements of the activity oriented language like Activity Diagrams widely used in modeling workflow networks. We have addressed the compensation requirements in such networks however we have not addressed the implementation requirements of such compensation from the model. This is another work that we are pursuing.

We have shown the use of Esterel based model checker *xeve* but which is presently not the state of the art. Augmenting the verification engine by using better techniques like SAT based verification would definitely allow to verify large models and involving nonlinear arithmetic. This is currently a quite active area in the formal verification community.

Another challenge is to bring the issues of service orchestration and choreography to model oriented designs. This we believe would be very useful in modeling large systems composed of communicating subsystems. Integration of SysML [69] in a formal framework of service orchestration and choreography will be a significant contribution to modeling community involved in software design at large. We are working towards such an integrated formalism to achieve the same via the reactive semantics. Furthermore, we plan as part of our future work to verify/validate the timing characteristics embedded in the request-response framework.

Our methods of deducing type correctness of C programs and validation of translation is based on deductive methodology and is not completely automatic. Automation of the proof is one which needs very careful study in terms of invariant generation and proofs using correct lemmas. This is particular if *grind* or *simplify* proof strategies fail and one needs to instantiate the correct lemma. This is in general a very active research area.

10.3 Main Contribution

The main contribution of the thesis is in the following:-

1. Formalization of semantics of Statecharts based on an imperative synchronous semantics of Esterel: We present complete algorithms for the translation of Statecharts into Esterel. This allows us to use other backend tools in the synchronous family for verification and high level code generation for implementation .

2. Extending the Statecharts to model communication: We present an extension in Statecharts to model communication through channels. The new language allows us to model communicating reactive systems. We present an alternate translation scheme to a language Promela used in a verification tool SPIN. We also show how such a specification can be realized in the imperative synchronous language ESTEREL .

3. Formalization of Activity Diagrams in Esterel: We present a reactive semantics of the various activity patterns using Esterel. We show that this semantics allows us to carry verification and code generation using tools used for family of synchronous languages [ICDCIT05].

4. Compensating Activity Diagrams: Although the Activity Diagrams can model most of the workflow patterns used in business process, however it cannot model failures in business processes. We show a possible extension of activity diagrams to model compensations required in such business process logic [JOT09].

5. Language to Model Choreography in Distributed Service Oriented Computing: Choreographs define the sequence of exchanging messages between two (or more) independent participants or processes by describing how they should cooperate. We show a formal language framework which integrates orchestration with scripting to abstract conversations leading to an effective modular specification of service choreography [APSCC08,ICWS08].

6. Type systems for C: Most of the model based design tools generate the implementation in C language. This is particularly true for tools used in embedded systems. However it is known that there is no proper type system for C and hence C programs may not be typesafe. We show an implementation of a system for checking types for a restricted class of C programs based on a type system implementation in PVS [SAFECOMP07].

7. Object Code Validation: The code generated from a model using a model compiler needs to be ultimately translated to a machine level code using a HLL compiler. It is known that compiler verification is an undecidable problem. We show how an alternate methodology based on Translation Validation can be used to verify fragments of HLL code which can be used for compiler validation [FTRTFT00].

References

[1] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[2] Rajeev Alur, Sampath Kannan, and Mihalis Yannakakis. Communicating hierarchical state machines. *Lecture Notes in Computer Science*, 1644:169–179, 1999.

[3] Stuart Anderson, Massimo Felici, and Bev Littlewood, editors. *Computer Safety, Reliability, and Security, 22nd International Conference, SAFECOMP 2003, Edinburgh, UK, September 23-26, 2003, Proceedings*, volume 2788 of *Lecture Notes in Computer Science*. Springer, 2003.

[4] Charles André. Synccharts: a visual representation of reactive behaviors. Technical Report RR 95–52, rev. RR (96–56), I3S, Sophia-Antipolis, France, Rev. April 1996.

[5] J. Augusto, M. Leuschel, M. Butler, and C. Ferreira. Using the extensible model checker XTL to verify stac business specifications. In *In Pre-proceedings of 3rd Workshop on Automated Verification of Critical Systems (AVoCS 2003)*, pages 253–266, 2003.

[6] R.K. Shyamasundar B. Rajan. Multiclock esterel: A reactive framework for asynchronous design. In *IPDPS '00: Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, page 201, Washington, DC, USA, 2000. IEEE Computer Society.

[7] Christel Baier and Holger Hermanns, editors. *CONCUR 2006 - Concurrency Theory, 17th International Conference, CONCUR 2006, Bonn, Germany, August 27-30, 2006, Proceedings*, volume 4137 of *Lecture Notes in Computer Science*. Springer, 2006.

[8] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. Rajamani, and A. Ustuner. Thorough static analysis of device drivers, 2006.

[9] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous programming with events and relations: the signal language and its semantics. *Sci. Comput. Program.*, 16(2):103–149, 1991.

[10] G. Berry, S. Ramesh, and R. K. Shyamasundar. Communicating reactive processes. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 85–98, New York, NY, USA, 1993. ACM.

[11] Gérard Berry. The effectiveness of synchronous languages for the development of safety-critical systems. Technical report, Esterel Technologies, 2003.

[12] Gérard Berry and Georges Gonthier. The esterel synchronous programming language: design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992.

[13] A. K. Bhattacharjee, S. D. Dhodapkar, Sanjit A. Seshia, and R. K. Shyamasundar. A graphical environment for the specification and verification of reactive systems. In Felici et al. [44], pages 431–444.

[14] A. K. Bhattacharjee, Gopa Sen, S. D. Dhodapkar, Kundapur Karunakar, Basant Rajan, and R. K. Shyamasundar. A system for object code validation. In Joseph [70], pages 152–169.

[15] A. K. Bhattacharjee and R. K. Shyamasundar. Validated code generation for activity diagrams. In Chakraborty [28], pages 508–521.

[16] A. K. Bhattacharjee and R. K. Shyamasundar. Choreography = orchestration with scripts + conversations. In *Proceedings of IEEE International Conference on Web Services, ICWS 2008 (To appear)*. IEEE, 2008.

[17] A. K. Bhattacharjee and R. K. Shyamasundar. Scriptorc : A specification language for web service choreography. In *Proceedings of IEEE International Asia Pacific Conference on Service Computing, APSCC 2008 (To appear)*. IEEE, 2008.

[18] A.K. Bhattacharjee, S.D. Dhodapkar, S. Seshia, and R.K. Shyamasundar. PERTS: An environment for specification and verification of reactive systems. *Reliability Engineering & Systems Safety Journal*, 71(2):299 –310, 2001.

[19] A.K. Bhattacharjee, S.D. Dhodapkar, and R.K. Shyamasundar. An environment for modeling communicating reactive systems. In *IT 2008, 1st IEEE International Conference on Information Technology*. IEEE Xplore, Digital Object Identifier 10.1109/INFTECH.2008.4621603, 2008.

[20] N. Bjorner, A. Browne, E. Chang, M. Col'on, A. Kapur, Z. Manna, H. Sipma, and T. Uribe. Step the stanford temporal prover educational release version, 1995.

[21] Amar Bouali. Xeve: an esterel verification environment (version v1.3).

[22] Douglass B.P. *Real Time UML Advances in the UML for Real-Time Systems*. Pearson Edition, 2004.

[23] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, 1984.

[24] Roberto Bruni, Hernán Melgratti, and Ugo Montanari. Theoretical foundations for compensations in flow composition languages. *SIGPLAN Not.*, 40(1):209–220, 2005.

[25] Michael J. Butler and Carla Ferreira. A process compensation language. In *IFM '00: Proceedings of the Second International Conference on Integrated Formal Methods*, pages 61–76, London, UK, 2000. Springer-Verlag.

[26] Michael J. Butler, C. A. R. Hoare, and Carla Ferreira. A trace semantics for long-running transactions. In *25 Years Communicating Sequential Processes*, pages 133–150, 2004.

[27] Luca Cardelli. Type systems. In Allen B. Tucker, editor, *The Handbook of Computer Science and Engineering*, Boca Raton, FL, 1997. CRC Press.

[28] Goutam Chakraborty, editor. *Distributed Computing and Internet Technology, Second International Conference, ICDCIT 2005, Bhubaneswar, India, December 22-24, 2005, Proceedings*, volume 3816 of *Lecture Notes in Computer Science*. Springer, 2005.

[29] Edward Y. Chang, Zohar Manna, and Amir Pnueli. Characterization of temporal property classes. In *ICALP '92: Proceedings of the 19th International Colloquium on Automata, Languages and Programming*, pages 474–486, London, UK, 1992. Springer-Verlag.

[30] A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new Symbolic Model Verifier. In N. Halbwachs and D. Peled, editors, *Proceedings Eleventh Conference on Computer-Aided Verification (CAV'99)*, number 1633 in Lecture Notes in Computer Science, pages 495–499, Trento, Italy, July 1999. Springer.

[31] Byron Cook and Andreas Podelski, editors. *Verification, Model Checking, and Abstract Interpretation, 8th International Conference, VMCAI 2007, Nice, France, January 14-16, 2007, Proceedings*, volume 4349 of *Lecture Notes in Computer Science*. Springer, 2007.

[32] Patrick Cousot. Proving the absence of run-time errors in safety-critical avionics code. In *EMSOFT '07: Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 7–9, New York, NY, USA, 2007. ACM.

[33] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, New York, NY, USA, 1977. ACM.

[34] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 269–282, New York, NY, USA, 1979. ACM.

[35] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, , and Mandayam Srivas. A tutorial introduction to pvs, April 1995.

[36] Werner Damm and David Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[37] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[38] Jin Song Dong, Yang Liu, Jun Sun, and Xian Zhang. Verification of computation orchestration via timed automata. In Zhiming Liu and Jifeng He, editors, *Formal Methods and Software Engineering, 8th International Conference on Formal Engineering Methods, ICFEM 2006*, volume 4260 of *Lecture Notes in Computer Science*, pages 226–245. Springer, 2006.

[39] Stephen A. Edwards. High-level synthesis from the synchronous language esterel. In *IWLS*, pages 401–406, 2002.

[40] Michael Emmi and Rupak Majumdar. Verifying compensating transactions. In Cook and Podelski [31], pages 29–43.

[41] Rik Eshuis. Symbolic model checking of uml activity diagrams. *ACM Trans. Softw. Eng. Methodol.*, 15(1):1–38, 2006.

[42] Rik Eshuis and Roel Wieringa. Verification support for workflow design with uml activity graphs. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 166–176, New York, NY, USA, 2002. ACM.

[43] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, /2002.

[44] Massimo Felici, Karama Kanoun, and Alberto Pasquini, editors. *Computer Safety, Reliability and Security, 18th International Conference, SAFECOMP'99, Toulouse, France, September, 1999, Proceedings*, volume 1698 of *Lecture Notes in Computer Science*. Springer, 1999.

[45] Jeffrey Fischer and Rupak Majumdar. Ensuring consistency in long running transactions. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 54–63, New York, NY, USA, 2007. ACM.

[46] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of web service compositions, 2003.

[47] Nissim Francez and Brent Hailpern. Script: A communication abstraction mechanism. In *PODC '83: Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 213–227, New York, NY, USA, 1983. ACM.

[48] X. Fu, T. Bultan, and J. Su. Analysis of interacting bpel web services. In *In Proc. of the 13th International World Wide Web Conference (WWW'04), USA, 2004. ACM Press.*, 2004.

[49] Hector Garcia-Molina and Kenneth Salem. Sagas. *SIGMOD Rec.*, 16(3):249–259, 1987.

[50] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[51] Nicolas Halbwachs. Synchronous programming of reactive systems. In *Computer Aided Verification*, pages 1–16, 1998.

[52] Anthony Hall. Correctness by construction: Integrating formality into a commercial development process. In *FME '02: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods - Getting IT Right*, pages 224–233, London, UK, 2002. Springer-Verlag.

[53] Anthony Hall and Roderick Chapman. Correctness by construction: Developing a commercial secure system. *IEEE Softw.*, 19(1):18–25, 2002.

[54] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, and a. Shtul-Trauring. Statemate: a working environment for the development of complex reactive systems. In *ICSE '88: Proceedings of the 10th international conference on Software engineering*, pages 396–406, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[55] D. Harel and R. Marelly. Specifying and executing behavioral requirements: The play-in/play-out approach. Technical Report MSC01-15, The Weizmann Institute of Science, 2001.

[56] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[57] David Harel and Amnon Naamad. The statemate semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293–333, 1996.

[58] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with blast. In *Lecture Notes in Computer Science*, volume 2648, pages 235–239. Springer, 2003.

[59] Tony Hoare. The verifying compiler: a grand challenge for computing research. Technical report, Microsoft,research.microsoft.com/ thoare, 2004.

[60] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[61] The MathWorks MATLAB Website: http://www.mathworks.com/. Matlab website: http://www.mathworks.com/.

[62] Polyspace Technologies Home Page http://www.polyspace.com, June 2007.

[63] Cornelis Huizing, Rob Gerth, and Willem P. de Roever. Modeling statecharts behaviour in a fully abstract way. In *CAAP '88: Proceedings of the 13th Colloquium on Trees in Algebra and Programming*, pages 271–294, London, UK, 1988. Springer-Verlag.

[64] IBM,BEA, Microsoft and Seibel, Available at http://www.ibm.com/developerworks/library /specification/ws-bpel/. *Business Process Execution Language for Web Services*, 1.1 edition, 2002.

[65] IEC60880. *Nuclear Power Plants I& C Systems Important to Safety Software aspects for Computer Based Systems Performing Category A Functions,* IEC60880*, Ed.2.* IEC, 2004.

[66] A. Iqbal, A. K. Bhattacharjee, S. D. Dhodapkar, and S. Ramesh. Visual modeling and verification of distributed reactive systems. In Anderson et al. [3], pages 22–34.

[67] Ajith K. John, Babita Sharma, A. K. Bhattacharjee, S. D. Dhodapkar, and S. Ramesh. Detection of runtime errors in misra c programs: A deductive approach. In Saglietti and Oster [101], pages 491–504.

[68] Barnes John. *High Integrity Ada: The SPARK Approach*. Addison-Wesley, 1997.

[69] Thomas Johnsona, Jonathan Jobe, Christiaan Paredis, and Roger Burkhart. Modeling continuous system dynamics in SYSML. In *Proceedings of the IMECE 2007*, 2007.

[70] Mathai Joseph, editor. *Formal Techniques in Real-Time and Fault-Tolerant Systems, 6th International Symposium, FTRTFT 2000, Pune, India, September 20-22, 2000, Proceedings*, volume 1926 of *Lecture Notes in Computer Science*. Springer, 2000.

[71] William Kahan. IEEE 754 standard for binary floating-point arithmetic. Technical report, IEEE, 1985.

[72] G. Kahn. The semantics of a simple language for parallel programming. *In Information Processing*, 74:993–998, 1977.

[73] David Kitchin, William R. Cook, and Jayadev Misra. A language for task orchestration and its semantic properties. In Baier and Hermanns [7], pages 477–491.

[74] Mariya Koshkina and Franck van Breugel. Modelling and verifying web service orchestration by means of the concurrency workbench. *SIGSOFT Softw. Eng. Notes*, 29(5):1–10, 2004.

[75] Edward A. Lee and Haiyang Zheng. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In *EMSOFT '07: Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 114–123, New York, NY, USA, 2007. ACM.

[76] Nancy G. Leveson, Mats Per Erik Heimdahl, Holly Hildreth, and Jon Damon Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, 1994.

[77] Jing Li, Huibiao Zhu, and Geguang Pu. Conformance validation between choreography and orchestration. In *First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering (TASE 2007)*, pages 473–482, June 2007.

[78] Jing Li, Huibiao Zhu, Geguang Pu, and Jifeng He. A formal model for compensable transactions. In *ICECCS '07: Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems*, pages 64–73, Washington, DC, USA, 2007. IEEE Computer Society.

[79] Peter Liggesmeyer, Klaus Pohl, and Michael Goedicke, editors. *Software Engineering 2005, Fachtagung des GI-Fachbereichs Softwaretechnik, 8.-11.3.2005 in Essen*, volume 64 of *LNI*. GI, 2005.

[80] Jeff Magee and Jeff Kramer. *Concurrency: State Models and Java Programs*. Wiley International, July 2006.

[81] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems: safety*. Springer-Verlag New York, Inc., New York, NY, USA, 1995.

[82] Keith Mantell. From UML to BPEL, model driven architecture in a web services world, http://www-128.ibm.com/developerworks/webservices/library/ws-uml2bpel, 2006.

[83] Florence Maraninchi and Yann Rémond. Mode-automata: About modes and states for reactive systems. *Lecture Notes in Computer Science*, 1381:185–195, 1998.

[84] E. Michael Maximilien and Munindar P. Singh. Toward web services interaction styles. In *Proceedings of 2nd IEEE International Conference on Services Computing (SCC)*. IEEE, July 2005.

[85] R. Milner. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[86] Jayadev Misra and William R. Cook. Computation orchestration: A basis for wide-area computing. *Journal of Software and Systems Modeling*, pages pp.83–110, May 2006.

[87] MISRA-C:2004. *Guidelines for the use of the C language in critical systems*. The Motor Industry Software Research Association, 2004.

[88] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Paris, January 1997.

[89] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, 2007.

[90] Object Management Group (OMG), Available at http://www.bpmi.org/. *Business Process Modeling Language (BPML)*, 1.0 edition, 2002.

[91] Object Management Group (OMG). OMG model driven architecture, version 1.0.1, 2004.

[92] Object Management Group (OMG). Unified modeling language : Superstructure, version 2.0, revised final adopted specification, 2004.

[93] Object Management Group (OMG). Business process modeling notation (BPMN) specification available at http://www.bpmn.org, 2006.

[94] S. Owre. PVS language reference, http://www.csl.sri.com, 2007.

[95] Sebastian Pavel, Jacques Noye, Pascal Poizat, and Jean-Claude Royer. Java implementation of a component model with explicit symbolic protocols, 2005.

[96] Chris Peltz. Web services orchestration and choreography. *IEEE Computer*, 36 (10):pp. 46–52, Oct., 2003.

[97] A. Pnueli, O. Shtrichman, and M. Siegel. Translation validation for synchronous languages. *Lecture Notes in Computer Science*, 1443:235–265, 1998.

[98] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. *Lecture Notes in Computer Science*, 1384:151+, 1998.

[99] Zongyan Qiu, Xiangpeng Zhao, Chao Cai, and Hongli Yang. Towards the theoretical foundation of choreography. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 973–982, New York, NY, USA, 2007. ACM.

[100] B. Rajan and R. Shyamasundar. An implementation of communicating reactive process, 1997. Intl. Conf. on Parallel and Distributed Computing and Networks, Singapore.

[101] Francesca Saglietti and Norbert Oster, editors. *Computer Safety, Reliability, and Security, 26th International Conference, SAFECOMP 2007, Nuremberg, Germany, September 18-21, 2007*, volume 4680 of *Lecture Notes in Computer Science*. Springer, 2007.

[102] Tim Schattkowsky and Wolfgang MÃijller. Model-based design of embedded systems. In *Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'04)*, pages pp. 121–128, 2004.

[103] Sanjit A. Seshia, R. K. Shyamasundar, A. K. Bhattacharjee, and S. D. Dhodapkar. A translation of statecharts to esterel. In Wing et al. [115], pages 983–1007.

[104] Munindar P. Singh and Michael N. Huhns. *Service-Oriented Computing, Semantics, Processes, Agents*. John Wiley & Sons, Ltd., 2005.

[105] Harald Störrle. Semantics of uml 2.0 activities with data-flow. In *German Software Engineering Conference*, 2005.

[106] Harald Störrle and Jan Hendrik Hausmann. Towards a formal semantics of uml 2.0 activities. In Liggesmeyer et al. [79], pages 117–128.

[107] Hoare T. Compensable transactions, May 2006. Slides Presented at UNI-IIST, Beijing.

[108] van der Aalst, W. ter Hofstede, and A.Kiepuszewski B.and Barros A. Workflow patterns. *Distributed and Parallel Databases*, 14(3):5–51, 2003.

[109] Russel N.and van der Aalst and W. ter Hofstede A .and Peta Wohed. On the suitability of uml 2.0 activity diagrams for business process modelling. *Proceedings of the Third Asia-Pacific Conference on Condeptual Modelling*, pages 95–104, 2006.

[110] W3C Working Draft, Available at http://www.w3.org/TR/wsci/. *Web Services Choreography Interface (WSCI) Language*, 1.0 edition, 2002.

[111] W3C Working Draft, Available at http://www.w3.org/TR/ws-cdl-10/. *Web Services Choreography Description Language*, 1.0 edition, 2004.

[112] W3C Working Draft, Available at http://www.w3.org/TR/wscl10/. *Web Services Conversation Language (WSCL)*, 1.0 edition, 2004.

[113] W3C Working Draft, Available at http://www.w3.org/TR/ws-chor-model/. *WS Choreography Model Overview*, 2004.

[114] I. Wehrman, D. Kitchin, W. Cook, and J. Misra. A timed semantics of orc. *Theoretical Computer Science*, 402(2-3):234–248, August 2008.

[115] Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors. *FM'99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 20-24, 1999, Proceedings, Volume II*, volume 1709 of *Lecture Notes in Computer Science*. Springer, 1999.

[116] Peter Y. H. Wong and Jeremy Gibbons. A Process-Algebraic Approach to Workflow Specification and Refinement. In *Proceedings of 6th International Symposium on Software Composition*, March 2007.

[117] W.L. Yeung. CSP based verification for web service orchestration and choreography. *Simulation*, 83(1):65–74, 2007.

Appendix A

Representation of Statecharts and Data Structures used in STATEST

Statecharts is a graphical language. However for processing the specification for translation, it is represented in textual form (LALR(1)). Any Statechart is built by using operators and primitive states. In Statecharts, there are two types of states namely the AND-type and the OR-type. Being in an AND-state means being in all of its immediate substates simultaneously. We call these immediate substates and their interior the orthogonal components of that AND-state. Being in an O-state means being in exactly one of its substates. The operator that build the interior of an AND-state is called an *AndChart* and the operator that build the inside of an OR-state is called *OrChart*. We use a Statification operator that builds the hierarchical structure of Statechart. The basic states are denoted by $Prim(I, O, A)$, where A is the name of a state, I and O a set of incoming and outgoing transitions. Prim Charts are nonterminal with one root state and is a Statechart, without incoming and outgoing transitions. The grammar (acceptable to YACC/Bison) specification of the textual representation is given below.

```
%{
#include <stdio.h>

#include <symtab.h>

#include <tree.h>
extern char *yytext;
extern int yylineno;
extern TREENODE_PTR rootPtr;


%}
%union
{
```

```
SYMTAB_PTR sym;
TREENODE_PTR tptr;
TRANSITION_PTR trPtr;
TRLIST_PTR trList;
EXPRESSION_PTR ePtr;
}
%start statechart
%token BASIC STAT OR AND IN EQ LG LT LE GT GE
%token<sym>IDENTIFIER
%type<tptr> statchart primchart orchart andchart
%type<sym> statelabel
%type<trList> input input_opt input_set output output_opt output_set
              history history_opt
%type<trPtr> transition
%type<ePtr> event condition action expr arr_expr logical_expr
              function state_prefix
%%
statechart : primchart

primchart : BASIC '(' input_opt ',' output_opt ',' statelabel ')'


|  statchart ',' orchart ',' input_opt ',' history_opt ')'

|  statchart ',' andchart ')'

;
statchart : STAT '(' BASIC '(' input_opt ',' output_opt ',' statelabel ')'

;
orchart : primchart

| OR '(' orchart ',' orchart ',' input_opt ',' output_opt ')'

;
andchart : primchart

| AND '(' andchart ',' andchart ',' input_opt ',' output_opt ')'

;
input_opt : empty

| input_set
```

```
;
output_opt : empty

| output_set

;
history_opt :
| history

;
input_set : '{' input '}'

;
input : transition

| transition ',' input

;
output_set : '{' output '}'

;
output : transition

| transition ',' output

;
transition : '[' statelabel ',' statelabel ',' event ',' condition ','
                   action ']'

;
statelabel : IDENTIFIER
;
event :
| logical_expr


;
condition :
| expr

;
action :
```

```
| expr

;
expr : IDENTIFIER '=' arr_expr

| '(' expr ')'

| logical_expr

| function

;
logical_expr : IDENTIFIER

| '(' logical_expr ')'


| IN statelabel

| IN state_prefix

| IDENTIFIER EQ logical_expr

| IDENTIFIER LG logical_expr

  | IDENTIFIER LT logical_expr

  | IDENTIFIER LE logical_expr

| IDENTIFIER GT logical_expr

| IDENTIFIER GE logical_expr

| '^' logical_expr %prec '^'

| '!' logical_expr %prec '^'

| logical_expr '|' logical_expr

| logical_expr '&' logical_expr

;
```

```
state_prefix : statelabel '.' statelabel


| statelabel '.' state_prefix

;
function : IDENTIFIER '(' expr ')'

;
arr_expr : IDENTIFIER

| IDENTIFIER '+' arr_expr

| IDENTIFIER '-' arr_expr

| IDENTIFIER '*' arr_expr

| IDENTIFIER '/' arr_expr

;
history : '(' IDENTIFIER ',' input_opt ')'

| '(' ',' ')'

;
empty : '{' '}'
;
```

Data Structures used in STATEST

The data structures used in the algorithms described in chapter 3 are given here

```
typedef enum {ENTR=1, EXIT,ENTR_EXIT,LOOP} TRCONDITION ;
typedef enum {TYPE1, TYPE2, TYPE3, TYPE4} TRTYPE ;
enum {BASIC_OP=0,OR_OP,AND_OP,STAT_OP};
enum {HISTORY=1,DEFAULT};


typedef struct hide {
        char *state; /* name of hide signal */
        struct hide *next; /* pointer to next hide signal in list*/
} *HIDE_LIST ,HIDE;
```

```
typedef struct succList
{
        void *data;
        int parentToChildType;
struct succList *next;
}LIST,*LIST_PTR;
typedef struct transition
{
        char *src;  /* source of transition */
        char *dest; /* destination of transition */
        TRTYPE trType; /* type of transition (TYPE1, TYPE2 etc.)*/
        int trSpFlag; /* */
        EXPRESSION_PTR event; /*Event associated with transition*/
        EXPRESSION_PTR condition;
        EXPRESSION_PTR action;
        HIDE_LIST noHide; /*Hide signal to be issued by this transition
             to hide other lower priority transitions*/
        LIST_PTR exitStateSet;
        LIST_PTR entryStateSet;
        int trId;
        char *lastExitedState;
}TRANSITION,*TRANSITION_PTR;

struct  transition_tag
{
        TRANSITION_PTR trPtr; /*Pointer to transition */
        TRCONDITION flag; /* indicated whether tr is ENTRY, EXIT or LOOP*/
        TRTYPE trType; /* type of transition */
        char *lastExitedState;/*Last state exited by the transition trPtr*/
        struct transition_tag *next;
};
typedef struct transition_list
{
        TRANSITION_PTR trPtr;
        struct transition_list *next;
}TRLIST,*TRLIST_PTR;

typedef struct tree_node
{
        char *name;
        int type;  /*operator */
        int histFlag;
        int childToParentType;
        int parentToChildType;
        int noChild;
        int levelInTree;
        TRLIST_PTR inputTrList;
        TRLIST_PTR outputTrList;
        struct tree_node *lchild;
        struct tree_node *rchild;
        LIST_PTR succPtr;
        LIST_PTR signalList;
        struct tree_node *parentPtr;
        struct transition_tag *trTag;
        HIDE_LIST hideLabels;

}TREENODE,*TREENODE_PTR;
typedef struct Signal{
        char *name;
        int type; /* 0= input, 1=output, 2=inputoutput */
}ESIG_DECL, *ESIG_DECL_PTR;
```

Appendix B

Brief Introduction to CSP, PVS and STeP

B.1 Brief Discussion on CSP

In CSP[23] the ultimate unit in the behaviour of a process is an event (conditions) which are regarded as instantaneous and A is the set of all events. The behaviour of a process upto some instant of time can be a record of events in which it has participated. The basic CSP processes are

$$P ::= STOP \mid SKIP \mid e \rightarrow P \mid c?x \rightarrow P \mid$$
$$P \Box Q \mid P \sqcap Q \mid P;\ Q \mid P \parallel_{\mathcal{X}} Q \mid P \backslash A \mid \mu x.P(x)$$

The process STOP can perform no events: it represents the end of a pattern of behaviour. The process SKIP can do nothing but terminate and the future behaviour is determined by the expression following the next sequential composition symbol. The process $e \rightarrow P$ ("e then P") is ready to perform the event e and if this event is performed, the future behaviour of this process is described by term P. The query symbol, ?, denotes a choice of events: the process $c?x \rightarrow P$ is ready to perform any event of the form c.x; if this process performs a particular event c.a, then x takes the value a for the rest of the current scope. The symbol $\Box$ denotes an external choice of behaviours. if x and y are distinct events $(x \rightarrow P \Box y \rightarrow Q)$ describes a process which initially engages in either of the event x or y. The notation $P \sqcap Q$ (P or Q) denotes a process which behaves either like P or like Q, where the choice is made internally and may represent run-time nondeterminism. P;Q denotes a process which initially behaves like P and upon successful termination of P behaves like Q. The parallel composition, $P \parallel Q$, specifies the process which behaves like the system composed of processes P and Q interacting in lock step synchronisation. The set of events that can occur only if performed simultaneously by both processes. In $P \parallel_X Q$, the execution of the activities in P and Q are synchronized over X. The hiding operator internalises sets of events: the expression $P \backslash A$ denotes a process that behaves exactly as P, except that events from the set A are no longer visible in the environment i.e. they may not be shared with, and do not require the cooperation of, other processes. A recursive process like a clock can be defined as $\mu x : (tick) \mid (tick \rightarrow x)$.

Traces play a central role in CSP in describing the behaviour of processes. A trace of the behaviour of a process is a finite sequence of symbols recording the events in which the process has engaged up to some momemnt in time. $traces[\,\bullet\,]$ is a semantic function which maps a CSP expression to its set of possible traces. The set of all such traces is defined by $traces[P] = \{s \in \mathcal{A}^* \mid \exists Q.P \xrightarrow{s} Q\}$. The general CSP process composition rules defined in terms of antecedent and consequent $\frac{A_1, A2, \cdots, A_n}{C}$ are defined below :

Termination	$\frac{}{SKIP \xrightarrow{\checkmark} STOP}$	
Prefix	$\frac{}{a \rightarrow P \xrightarrow{a} P}$	
Choice	$\frac{P \xrightarrow{a} P'}{P \Box Q \xrightarrow{a} P'}$	$\frac{Q \xrightarrow{a} Q'}{P \Box Q \xrightarrow{a} Q'}$
Sequence	$\frac{P \xrightarrow{a} P' \wedge a \neq \checkmark}{P;\, Q \xrightarrow{a} P';\, Q}$	$\frac{P \xrightarrow{\checkmark} 0 \wedge Q \xrightarrow{a} Q'}{P;\, Q \xrightarrow{a} Q'}$
Interleaving Parallel	$\frac{P \xrightarrow{a} P' \wedge a \notin \mathcal{A}}{P \Vert_{\mathcal{A}} Q \xrightarrow{a} P' \Vert_{\mathcal{A}} Q}$	$\frac{Q \xrightarrow{a} Q' \wedge a \notin \mathcal{A}}{P \Vert_{\mathcal{A}} Q \xrightarrow{a} P \Vert_{\mathcal{A}} Q'}$
Synch. Rule	$\frac{P \xrightarrow{a} P' \wedge Q \xrightarrow{a} Q' \wedge a \in \mathcal{A}}{P \Vert_{\mathcal{A}} Q \xrightarrow{a} P' \Vert_{\mathcal{A}} Q'}$	

B.2 A Brief Discussion on PVS

Prototype Verification System(PVS) [35] is a verification system supporting an interactive environment for writing formal specifications and checking formal proofs. It provides an expressive language that augments classical higher order logic with a sophisticated type system containing predicate subtypes, dependent types, and with parameterised theories and a mechanism for defining abstract datatypes such as lists and trees.The standard PVS types include numbers (reals, rationals, integers, naturals). The combination of features in the PVS typesystem is very convenient for specification, but it makes typechecking undecidable. The PVS typechecker copes with this undecidability by generating proof obligations for the PVS theorem prover. Most such proof obligations can be discharged automatically. PVS has a powerful interactive theorem prover/proof checker. The basic deductive steps in PVS are large compared with many other systems: there are atomic commands for induction, quantifier reasoning, automatic conditional rewriting, simplification using arithmetic and equality decision procedures and type information, and propositional simplification using binary decision diagrams. The PVS proof checker manages the proof construction process by prompting the user for a suitable command for a given subgoal. The execution of the given command can either generate further subgoals or complete a subgoal and move the control over to the next subgoal in a proof. User-defined proof strategies can be used to enhance the automation in the proof checker. Model-checking capabilities used for automatically verifying temporal properties of finite-state systems have recently been integrated into PVS. PVS's automation suffices to prove many straightforward results automatically; for hard proofs, the automation takes care of the details and frees the user to concentrate on directing the key steps.

B.3 A Brief Discussion on STeP

The Stanford Temporal Prover (STeP) [20] is a system developed to support the computer-aided formal verification of concurrent and reactive systems based on temporal specifications. Unlike systems based on model-checking, STeP is not restricted to finite-state systems. It combines model checking and deductive methods to allow the verification of a broad class of systems, including programs with infinite data domains, N -process programs, and N -component circuit designs, for arbitrary N . In short, STeP has been designed with the objective of combining the expressiveness of deductive methods with the simplicity of model checking. The verification process is for the most part automatic. User interaction occurs mostly at the highest, most intuitive level, primarily through a graphical proof language of verification diagrams. Efficient simplification methods, decision procedures, and invariant generation techniques are then invoked automatically to prove resulting first-order verification conditions with minimal assistance.First, in addition to the textual language of temporal logic, the system supports a structured visual language of verification diagrams for guiding, organizing, and displaying proofs. Verification diagrams allow the user to construct proofs hierarchically, starting from a high-level, intuitive proof sketch and proceeding incrementally, as necessary, through layers of greater detail.

Second, the system implements powerful techniques for automatic invariant generation. Deductive verification in the temporal framework almost always relies on finding, for a given program and specification, suitably strong (inductive) invariants and intermediate assertions. The user can typically provide an intuitive, highlevel invariant, from which the system derives stronger, more detailed, top-down invariants. Simultaneously, bottom-up invariants are generated automatically by analyzing the program text. By combining these two methods, the system can often deduce sufficiently detailed invariants to carry through the entire verification process.

Finally, the system provides an integrated suite of simplifications and decision procedures for automatically checking the validity of a large class of first-order and temporal formulas. This degree of automated deduction is sufficient to handle most of the verification conditions that arise during the course of deductive verification– and the few conditions that are not solved automatically typically correspond to the critical steps of manually constructed proofs, where the user is most able to provide guidance.

The basic input to STeP is an SPL program P and a temporal logic formula which expresses the property of P to be verified. The SPL program is modeled as a fair transition system S. Even though SPL can be used to describe both software and hardware systems, STeP is not restricted to SPL, and can be used to verify any system that can be modeled as a fair transition system.

Papers Published out of this Ph.D Thesis Work

Journal Publications

1. (JOT) A. K. Bhattacharjee, R. K. Shyamasundar, *Compensating Activity Diagrams: A Notation to Model Business Processes*, Accepted for publication in Journal of Object Technology, ETH Zurich, ISSN 1660-1769 To appear in January-February Issue, 2009.
2. (RESS01) A. K. Bhattacharjee, S. D. Dhodapkar , S. Seshia and R. K. Shyamasundar, *PERTS: an environment for specification and verification of reactive systems*, Elsevier Journal of Reliability Engineering and System Safety, 71(3), 2001,pp 299-310 ISSN: 0951-8320.

Peer Reviewed International Conference Publications:

1. (APSCC2008) A.K. Bhattacharjee and R.K. Shyamasundar *ScriptOrc : A Specification Language for Web Service Choreography*, Accepted for publication in the proceedings of the IEEE International Conference on Asia-Pacific Services Computing Conference (IEEE APSCC 2008) Yilan, Taiwan, 2008.
2. (ICWS2008) A.K. Bhattacharjee and R.K. Shyamasundar *Choreography = Orchestration with Scripts + Conversations*, IEEE International Conference on Web Services (IEEE ICWS 2008), Beijing, 2008.
3. (IT2008) A. K. Bhattacharjee, S. D. Dhodapkar, R.K. Shyamasundar, *An Environment for Modeling Communicating Reactive Systems*, Proceedings of International IEEE Conference on Information Technology, Gdansk University of Technology, Poland, May 18 - 21, 2008
4. (SAFECOMP07) Ajith K. John, Babita Sharma, A. K. Bhattacharjee, S. D. Dhodapkar, S. Ramesh,*Detection of Runtime Errors in MISRA C Programs: A Deductive Approach*, Computer Safety, Reliability, and Security, 26th International Conference, SAFECOMP 2007, Nuremberg, Germany, September 18-21, 2007. Lecture Notes in Computer Science 4680 Springer 2007, ISBN 978-3-540-75100-7.
5. (ICDCIT05) A. K. Bhattacharjee, R. K. Shyamasundar, *Validated Code Generation for Activity Diagrams*, Distributed Computing and Internet Technology, Second International Conference, ICDCIT 2005, Bhubaneswar, India, December 22-24, 2005, Proceedings. Lecture Notes in Computer Science 3816 Springer 2005 ISBN 3-540-30999-3.

6. (SAFECOMP03) A. Iqbal, A. K. Bhattacharjee, S. D. Dhodapkar, S. Ramesh, *Visual Modeling and Verification of Distributed Reactive Systems*, Computer Safety, Reliability, and Security, 22nd International Conference, SAFECOMP 2003, Edinburgh, UK, September 23-26, 2003, Proceedings. Lecture Notes in Computer Science 2788 Springer 2003, ISBN 3-540-20126-2.

7. (FTRTFT00) A. K. Bhattacharjee, Gopa Sen, S. D. Dhodapkar, Kundapur Karunakar, Basant Rajan, R. K. Shyamasundar,*A System for Object Code Validation*, Formal Techniques in Real-Time and Fault-Tolerant Systems, 6th International Symposium, FTRTFT 2000, Pune, India, September 20-22, 2000, Proceedings. Lecture Notes in Computer Science 1926 Springer 2000, ISBN 3-540-41055-4

Biography of Candidate

Name	Anup Kumar Bhattacharjee
Qualification	M.Tech(Computer Science), IIT Kharagpur
	B.E (Electrical Engineering), NIT, Silchar, Assam
	1 year Course in Nuclear Engineering at BARC (1987-1988)
Employment	Scientist, Bhabha Atomic Research Centre(BARC), Mumbai
Area of Work	Software Engineering, Formal Specification & Verification,
	Model based Software Design, Design of Reactive Systems
	Static Analysis

He is working in Software Reliability Section, BARC and has been closely associated with Mr. S.D. Dhodapkar who is known for his contribution toward methodologies for *High Integrity Safety-Critical System* Design. He has contributed significantly toward the development of static analyzers for C and assembly languages in BARC. During the last few years he has been working toward inducting formal methods for software developments in BARC. He has been closely associated with the research groups at Centre for Formal Design and Verification of Software (CFDVS) at IIT Bombay and School of Technology and Computer Science, TIFR. He has also worked in many collaborative research projects with other national laboratories like ISRO and DRDO. He was a visiting scholar in 2004 at the University of Trento, Italy under an Indo-Italian academic program.

Biography of Supervisor

Name	Prof. R.K. Shyamasundar
Qualification	Ph.D (Computer Science and Automation),IISc, Bangalore
	M.E. (Electrical Engg.) IISc, Bangalore
	B.E. (Electrical Engg.), University of Mysore
Area of Work	Real-time Distributed Systems, Programming Languages,
	Logic Programming, Reactive systems & Formal Methods

He joined the National Centre for Software Development and Computing Techniques established at the Tata Institute of Fundamental Research (TIFR) under United Nations Program. He did his post-doctoral work during 1978-1979 as an International Research Fellow at Eindhoven Technological University, Eindhoven, Netherlands under the famed Professor Dr. Edsgar W Dijkstra. On his return he started the Theoretical Computer Science Group at TIFR covering areas of concurrency, real-time programming, specification and verification of software etc. He was the first Dean of the School of Technology and Computer Science at the Tata Institute of Fundamental Research. He had various assignments at IBM TJ Watson Research center, Eindhoven University of Technology, The Netherlands, State University of Utrecht, the Netherlands, Pennsylvania State University, University of Illinois at Urbana, University of California, San Diego, ENSMP Sophia Antipolis, IRISA, Rennes, Verimag Grenoble Max Planck

Institute for Computer Science at Saarbrucken etc. He has more than 200 publications and several patents in US and India. Thirty students have done their Ph.D. under his guidance in India and abroad.

He has served as a consultant to Esprit projects at The Netherlands and several industries in India. He has worked towards setting centres of excellence such as Centre for Formal Design and Verification of Systems (CFDVS), IIT Bombay (Center set up from BRNS with participation from TIFR, BARC, and IIT Bombay), Cyber Security Education centers under DIT, Govt of India, Discrete Mathematics and Theoretical computer Science Centers under support from DST. He has contributed significantly in guiding the international co-operations such as Indo- French, Indo-Italian and Indo-US in the research areas of computer science.

He is a Fellow of the Indian Academy of Sciences, Fellow of the Indian National Science Academy, Fellow of the National Academy of Sciences,, Fellow of the Indian National Academy of Engineering and Fellow IEEE (USA). He is member of IEEE standardization committee on Esterel.