Doctoral Thesis

Equivalence checking for asynchronous software

Author[s]:
Haroud, Malek

Publication Date:
2005

Permanent Link:
https://doi.org/10.3929/ethz-a-005124898

Rights / License:
In Copyright - Non-Commercial Use Permitted
Equivalence Checking for Asynchronous Software

A dissertation submitted to the
Swiss Federal Institute of Technology Zürich
ETH Zürich

for the degree of
Doctor of Technical Sciences

presented by
Malek Haroud
Dipl. Elektrotek-Ing. EPFL
born July 20, 1972
citizen of Plaffein FR,
Switzerland

accepted on the recommendation of
Prof. Dr. Armin Biere, examiner
Prof. Dr. Lothar Thiele, co-examiner

October 2005
To my parents, to my wife, to my daughters, to all of you!
Acknowledgments

I take this opportunity to express my deep appreciation to my thesis supervisor Armin Biere without whose guidance, encouragement and criticism this work would not have appeared. Fortunately, Armin insisted from the very beginning of this thesis on setting a precise goal and a practical orientation as well. His trust in me was a constant source of motivation. I am also indebted to STMicroelectronics management namely Philippe Rouzet, Gerard Fargere and Dominique Bellot for supporting and sponsoring this research. I have benefited from conversations with many people in STMicroelectronics particularly Christophe Devaucelle and Regis Cattenoz. I would like to thank Professor Lothar Thiele who have accepted to be part of the defense committee. Finally, I would like to express my gratitude to my family for their love, support and encouragement especially for my wife Faiza whose patience and sacrifice were admirable throughout my studies.

Malek Haroud

Chavannes-sur-Moudon
June 5th 2005
Table of Contents

1 Introduction
   1.1 Model-based software engineering ........................................... 17
   1.2 Synchrony versus Asynchrony ................................................. 18
   1.3 Golden reference model ....................................................... 20
   1.4 Problem statement ............................................................. 20
   1.5 Alternative solutions .......................................................... 21
      1.5.1 Testing ................................................................. 21
      1.5.2 Formal compiler verification .......................................... 23
      1.5.3 Translation validation ................................................. 24
   1.6 Background ................................................................. 24
   1.7 Thesis contribution ......................................................... 25
   1.8 Significance ................................................................. 26

2 Related work
   2.1 RTL Versus Netlist Equivalence Checking .................................... 27
      2.1.1 State machine equivalence ........................................... 29
      2.1.2 Product machine ....................................................... 29
      2.1.3 Register correspondence ............................................. 30
   2.2 Assembly code optimization translation validation ........................... 31
   2.3 Signal to C translation validation ........................................... 33
      2.3.1 Verification conditions .............................................. 35
   2.4 Behavioral consistency of C and RTL Verilog ................................. 37

3 SDL and its runtime structure ...................................................... 39
   3.1 Overview ................................................................. 39
   3.2 SDL abstraction levels ....................................................... 39
      3.2.1 Structure .............................................................. 39
      3.2.2 Interfaces ............................................................. 40
      3.2.3 Data ................................................................. 41
      3.2.4 Behavior ............................................................. 41
   3.3 Layout mechanisms ........................................................... 43
      3.3.1 State multi-appearance .............................................. 43
      3.3.2 State shorthand notations .......................................... 43
      3.3.3 Input shorthand notations .......................................... 43
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4</td>
<td>Communication mechanisms</td>
<td>43</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Outputs</td>
<td>43</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Queuing signals handling</td>
<td>44</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Save</td>
<td>44</td>
</tr>
<tr>
<td>3.4.4</td>
<td>Timers</td>
<td>45</td>
</tr>
<tr>
<td>3.4.5</td>
<td>Enabling condition</td>
<td>45</td>
</tr>
<tr>
<td>3.4.6</td>
<td>Continuous signal</td>
<td>46</td>
</tr>
<tr>
<td>3.4.7</td>
<td>Import and export</td>
<td>46</td>
</tr>
<tr>
<td>3.5</td>
<td>Runtime structure</td>
<td>47</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Server model</td>
<td>47</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Activity thread model</td>
<td>48</td>
</tr>
<tr>
<td>4</td>
<td>The intermediate representation</td>
<td>51</td>
</tr>
<tr>
<td>4.1</td>
<td>Asynchronous Process Network</td>
<td>51</td>
</tr>
<tr>
<td>4.2</td>
<td>APN syntax</td>
<td>52</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Prefix and Choice operators</td>
<td>53</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Reachability predicate</td>
<td>53</td>
</tr>
<tr>
<td>4.3</td>
<td>APN Semantics</td>
<td>54</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Operational semantics</td>
<td>54</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Input/Output semantics</td>
<td>55</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Background transition semantics</td>
<td>56</td>
</tr>
<tr>
<td>4.3.4</td>
<td>Guarded input semantics</td>
<td>56</td>
</tr>
<tr>
<td>4.3.5</td>
<td>Parallel composition operator semantics</td>
<td>57</td>
</tr>
<tr>
<td>5</td>
<td>APN Equivalence Checking</td>
<td>61</td>
</tr>
<tr>
<td>5.1</td>
<td>Cutpoints</td>
<td>61</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Inductive assertions revisited</td>
<td>61</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Observation function</td>
<td>62</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Setting-up the invariant</td>
<td>63</td>
</tr>
<tr>
<td>5.1.4</td>
<td>Cutpoints classes</td>
<td>64</td>
</tr>
<tr>
<td>5.2</td>
<td>Process-level equivalence</td>
<td>65</td>
</tr>
<tr>
<td>5.3</td>
<td>Parallel-level equivalence</td>
<td>66</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Server model</td>
<td>66</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Server model versus Activity thread model</td>
<td>67</td>
</tr>
<tr>
<td>6</td>
<td>Compiling to APN</td>
<td>69</td>
</tr>
<tr>
<td>6.1</td>
<td>Compilation as a rewrite system</td>
<td>69</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Compiling C FSM code to APN</td>
<td>72</td>
</tr>
<tr>
<td>6.2</td>
<td>Control flow graph construction</td>
<td>73</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Threading the AST</td>
<td>73</td>
</tr>
<tr>
<td>6.3</td>
<td>Micro-transitions generation</td>
<td>76</td>
</tr>
<tr>
<td>6.4</td>
<td>Equivalence condition generation</td>
<td>77</td>
</tr>
</tbody>
</table>
# Table of Contents

7 Experiments ................................................................. 81
  7.1 SCEC Flow ........................................................................ 81
  7.2 Experiments ................................................................. 82
    7.2.1 The WiFi MAC station case study ................................. 82
    7.2.2 The MIPv6 protocol case study ................................... 83
  7.3 Finding bugs in the compiler ............................................ 84

8 Conclusion ........................................................................ 87

9 Annex: Hypotheses ............................................................ 89
  9.1 SDL sub-set restriction .................................................... 89
  9.2 SDL compiler ............................................................... 90
  9.3 Scheduling ................................................................. 90
## List of Figures

1.1 Evolution of system development ........................................... 18  
1.2 Synchrony versus Asynchrony ............................................. 19  
1.3 Design flow ................................................................. 20  
1.4 Back-to-back testing ....................................................... 21  
1.5 Compiler verification ....................................................... 23  
1.6 Translation validation infrastructure .................................... 24  
2.1 Hardware design flow ....................................................... 28  
2.2 Product machine ............................................................. 29  
2.3 Register correspondence .................................................... 31  
2.4 C versus optimized assembly ............................................... 32  
2.5 Assembly code translation validation ..................................... 33  
2.6 State correspondence ....................................................... 35  
2.7 Behavioral versus RTL translation validation ......................... 37  
3.1 Structure of an SDL system ............................................... 40  
3.2 (a) Process symbols (b) Example of SDL process .................... 42  
3.3 Queuing signal handling ................................................... 44  
3.4 Queuing with `Save' ....................................................... 45  
3.5 Enabling and continuous .................................................. 46  
3.6 Continuous signal .......................................................... 46  
3.7 Process descriptor used by the CAdvanced code generator .......... 48  
3.8 Process descriptor used by the CMicro code generator ............. 49  
4.1 Macro versus micro transitions ........................................... 54  
4.2 A macro-transition embedding two micro-transitions ............... 54  
4.3 Operational semantics: Conventions ..................................... 55  
4.4 Implicit consumption ...................................................... 56  
4.5 Background transition ..................................................... 56  
4.6 Server model versus Activity Thread model ........................... 58  
5.1 Computational induction ................................................... 62  
5.2 Observational equivalence ............................................... 63  
5.3 Cutpoints correspondence ............................................... 63  
5.4 Modular verification using cutpoints ................................... 64  
5.5 Cutpoint classes ........................................................... 65
List of Figures

5.6  Connection cutpoint ................................................................. 65
5.7  Parallel composition operator: server model $\psi$ .................................. 67
5.8  Server model $\psi$ versus Activity thread model $\omega$ .......................... 68

6.1  Structured variable flattening implementation .................................. 71
6.2  (a) SDL process P1 (b) Process descriptor’s C data structure ............... 73
6.3  C functions implementing process P1 .............................................. 74
6.4  Control flow graph of binary expression .......................................... 75
6.5  Control flow graph of a conditional statement .................................... 75

7.1  Flow ......................................................................................... 81
7.2  802.11 IEEE MAC for a station ......................................................... 82
7.3  macro and micro-transitions to be matched ......................................... 83
7.4  SCEC execution time on the MAC layer .............................................. 84
7.5  (a) MIPv6 System (b) Access Router block ......................................... 85
7.6  (a) Macro and micro-transitions statistics (b) SCEC execution time ....... 85
# List of Tables

1.1 Conformance to standards: Test-suites ........................................... 25

2.1 Intermediate representation ......................................................... 32
2.2 Signal primitives ........................................................................ 34
2.3 Signal semantics ......................................................................... 35
2.4 Inductive argument for proving the refinement relation .................... 36

4.1 Choice and prefix operators ......................................................... 53
4.2 Axioms of the prefix and choice operators ..................................... 53
4.3 Input and output semantics .......................................................... 55
4.4 Guarded input semantics ............................................................... 57

6.1 Structured variable flattening ....................................................... 70
6.2 Deref/Addr elimination ................................................................. 71
6.3 signal output reconstruction ....................................................... 72
6.4 (a) yStaH interpretation (b) State-event transition matrix .................. 74
Abstract

Model-driven engineering (MDE) [59] aims at describing and implementing complex systems starting from abstract models as opposed to the traditional code-centric methods. It is based on the idea of platform independent models and platform description models that can be realized using a variety of middleware and programming languages into platform specific models. Therefore, the MDE paradigm is more adapted than traditional approaches to cope with today’s systems complexity.

The specification and description language SDL belongs to the model-centric paradigm and has been successfully used at the telecommunication standardization bodies and in the industry as well. However, if we compare it to the code-centric approach, the time and effort invested in developing a platform independent model needs to be compensated. Consequently, model-to-code transformers are provided to derive automatically the implementation from the models. Ensuring the correctness and the reliability of these transformers is the key point that is addressed in this work.

This thesis addresses the problem of equivalence checking in the context of asynchronous software from a practitioner point of view. Our research builds on previous and ongoing work on compiler verification and translation validation. But, rather than simply focusing on the synchronous or purely sequential programs, it addresses the parallel asynchronous composition aspects as well.

In this work, we provide a verification methodology in which developers can automatically verify the refinement of an abstract reference model into the final target code. In such a flow, there is absolutely no need to re-validate again the target code obtained manually or by using a compiler. Without loss of generality, we treated the case where SDL is the source language and C the target implementation language.

In order to enable equivalence checking, both abstract and concrete programs need first to be translated into an intermediate form. The design of this intermediate representation was an important challenge of this thesis. Following the principle of union of concepts, we proposed an intermediate representation which coerces relevant aspects of both languages. This intermediate representation was defined in terms of its syntax and operational semantics.
Once both programs are translated into the intermediate representation, correspondence links are setup between the abstract and the concrete programs thanks to the concept of cutpoints. Their usage is a key concept in our method since they allow to subdivide the verification problem into smaller sub-problems. Cutpoints are also used to break the loops as in Floyd’s method [57].

Under such correspondence, we formalized the notion of equivalence by providing an inductive argument based on bisimulation. The sequential case was addressed together with two cases of the asynchronous parallel composition namely the server model and the activity thread model.

Our equivalence checking method was motivated by pragmatics. We insisted therefore to achieve a fully automatic method to check the equivalence systematically at each run of the compiler. Since program equivalence is undecidable for most interesting cases, we have imposed several simplifying assumptions under which the equivalence checking becomes algorithmically solvable and scalable as well. The main assumptions that underly this research concern the restriction to a subset of SDL, the compilation scheme and the scheduling regime.

To demonstrate the effectiveness of our approach, we have built a tool called SCEC. This tool is written in ANSI C. Flex and Bison specifications were also used to produce the scanners and the parsers for C and SDL. A WxWidgets based graphical user interface has been developed in order to browse the intermediate representation of both programs. Starting from the syntax tree, SCEC can record all the transformations that are applied on the trees up to the final normal form. Proof-obligations are generated automatically and are discharged using the ICS tool (i.e.; Integrated Canonizer and Solver ICS[30]). Currently, SCEC handles the Telelogic CAdvanced compiler only. Rewrite rules are hard-coded inside the tool. We claim that using a term rewriting engine [82], other code generators can be easily retargeted.

The experiments part contains two case studies: the IEEE 802.11 MAC layer and the MIPv6 protocol. The first case study represents a typical industrial software project. SCEC verified the C source code against its associated SDL specification in about one minute. In addition, these two case studies showed that the selected SDL sub-set is sufficiently expressive to describe real-world protocols. One major outcome is that during the verification process of the MAC layer, a subtle bug in the CAdvanced compiler was discovered. This result is very encouraging and shows that the development of such equivalence checker does pay off. We believe that the equivalence checking concept can be generalized to Domain-specific languages whose user base of their compilers are necessarily smaller and therefore less robust in comparison to main stream compilers.
Résumé

L’ingénierie dirigée par les modèles (IDM) [59] a pour objectif de décrire et de mettre en œuvre des systèmes complexes en partant de modèles abstraits par opposition à l’approche plus traditionnelle ciblant directement un code proche de la réalisation finale. L’idée de base repose sur une séparation entre des modèles indépendants des plateformes cibles et de ceux présents dans les produits finaux. Par conséquence, le paradigme IDM permet une meilleure maîtrise de la complexité des systèmes actuels que l’approche traditionnelle.

Le langage de description et de spécification LDS suit le concept IDM et a été intégré avec succès aussi bien par les organismes de standardisation que par l’industrie. L’effort de développement d’un modèle dans un langage comme LDS ne pourra se compenser que par l’utilisation de compilateurs dédiés permettant d’en dériver une réalisation efficace. La problématique centrale de cette thèse est de garantir la justesse de la transformation du modèle abstrait vers sa réalisation sur cible. En d’autres termes, il s’agit de s’assurer que le compilateur n’a pas introduit d’erreurs. Cette thèse traite du problème de la vérification d’équivalence dans le contexte de logiciels asynchrones sur un plan pratique. Notre recherche repose sur les travaux liés à la vérification de compilateurs mais ne se focalise par seulement sur des programmes synchrones ou séquentiels mais s’intéresse également à la composition parallèle et asynchrone.

Ce travail met à disposition une méthodologie de vérification qui permet aux développeurs de vérifier automatiquement la justesse du raffinement du modèle abstrait vers le code cible. Sans restriction du champ d’application, nous nous sommes intéressés au cas où le programme source est exprimé en LDS et traduit en code ANSI C. Afin de permettre la vérification d’équivalence, le modèle abstrait ainsi que le code cible correspondant doivent être exprimé dans une représentation intermédiaire commune. La conception d’une telle représentation a été le défi majeur relevé par ce travail. Nous avons opté pour la définition d’un langage intermédiaire qui unifie les types d’énoncés de base des deux langages LDS et ANSI C. Ce langage intermédiaire a été formellement exprimé tant du point de vue syntaxique que de sa sémantique opérationnelle.

Une fois que les deux programmes LDS et C sont traduits vers la représentation intermédiaire, des points de correspondance sont introduits grâce au principe des points de coupures. Leur utilisation permet de décomposer le problème de vérification en un ensemble de sous-problèmes plus gérables. Les points de coupures sont également utilisés...
pour ouvrir les boucles d’itération comme dans la méthode de Floyd [57]. Par ailleurs, nous avons formalisé la notion d’équivalence en utilisant un argument inductif basé sur la relation de bisimulation. Le cas séquentiel a été traité ainsi que deux cas de la composition parallèle asynchrone à savoir le modèle du serveur et celui de la serialisation des tâches.

Notre méthode de vérification d’équivalence répond à un problème concret auquel fait face l’industrie. Nous avons donc insisté sur la réalisation d’un logiciel qui soit complètement automatique et qui permet de vérifier chaque compilation systématiquement. Bien entendu, selon la théorie de la calculabilité, il n’existe aucun algorithme pouvant d’une manière générale prouver l’équivalence de deux programmes. Néanmoins, nous nous sommes attelés à trouver les hypothèses supplémentaires qui permettent de fournir un algorithme de vérification qui puisse traiter des problèmes de taille industrielle. Ces hypothèses font partie de trois catégories: la restriction du langage LDS à un sous-ensemble, le modèle de compilation ainsi que la méthode d’ordonnancement des processus LDS.

Un prototype appelé SCEC a été développé dans le cadre de cette thèse. Ce prototype est écrit en ANSI C et se base sur des spécifications Flex et Bison pour l’analyse lexicale et syntaxique des deux langages LDS et ANSI C. Une interface graphique permet de visualiser toutes les étapes intermédiaires de transformation des deux programmes vers la forme commune. Les obligations de preuves sont levées grâce à ICS [30]. Actuellement, SCEC vérifie les compilations réalisées par CAdvanced de Telelogic. Les règles de réécritures sont pour l’instant figées mais on pourra envisager dans le futur, l’utilisation d’un interpréteur de règles de réécriture [82] afin de cibler d’une manière plus flexible d’autres compilateurs.

La partie expérimentale comprend deux études de cas: la couche MAC 802.11 ainsi que le protocole MIPv6. Le premier cas, représente une situation typique d’un projet industriel. SCEC a pu vérifier la totalité du LDS et du C en moins d’une minute. De surcroît, ces deux études de cas montrent bien que le sous-ensemble du langage LDS est réaliste et suffisamment expressif. SCEC a pu détecter une erreur très subtile du compilateur CAdvanced. Ceci prouve d’une part, l’efficacité de notre approche et d’autre part, que le développement de ce type de vérificateur en vaut la peine. Nous sommes persuadés que notre concept de preuve d’équivalence peut se généraliser aux langages dédiés, dont les compilateurs sont nécessairement moins robustes, que les compilateurs classiques.
Introduction

*Model-driven engineering* aims at describing and implementing complex systems starting from abstract models as opposed to the traditional code-centric methods. It is based on the idea of platform independent models (PIM) and platform description models (PDM) that can be realized using a variety of middleware and programming languages into platform specific models (PSM).

This introductory chapter presents the benefits of the *model-driven engineering* methods over traditional approaches to cope with today’s systems complexity. The specification and description language SDL belongs to the model-centric paradigm and has been successfully used in the telecommunication standardization bodies and in the industry as well. However, if we compare it to the code-centric approach, the time and effort invested in developing a platform independent model needs to be compensated. Consequently, model-to-code transformers are provided to derive automatically the implementation from the models. Ensuring the correctness and reliability of these transformers is the key point that is addressed in this work. We discuss first the different alternatives that are available to alleviate this problem and finally argue in favor to the translation validation approach.

1.1 Model-based software engineering

The development of programming languages exhibits a clear direction towards higher levels of abstraction. This development started from assembly languages, then evolved to procedural languages[66], followed by object-oriented languages[52] and recently to component-oriented languages[61]. Actually, programs written in high level languages are much easier to read, less error prone and less expensive to maintain than similar programs written at a lower level of abstraction. As far as costs are concerned, software maintenance [85] consumes up to 75% of the total cost and is by far the biggest concern of those who develop and maintain commercial software product [35]. In addition, the necessity for reusing software components in systems becomes significant due to the ever increasing software complexity, product diversification and market pressure.

In order to facilitate the reuse [29] of components top down approaches classified as *Model Based Software Engineering* emerged. UML and SDL [27, 32] fall in this category. In such *model-centric* approach, programs are developed in a high-level graphi-
Fig. 1.1: Evolution of system development

cal languages and are extensively simulated and validated until they reach the status of golden reference model. Implementation is derived from the reference model by performing source-to-source code transformation [19, 42, 53, 12]. The target code is usually C or C++ [39]. In other words, the model based approach is attempting to bridge a large semantic gap between abstract models and concrete target code through the use of code generators.

An equally important requirement to manage today’s software complexity is to renounce to the concept of single locus of control. Apart from hard real time reactive system, software must shift from synchrony to asynchrony. Synchronous software is too restrictive for building large systems. Synchrony creates evil control dependencies between separate parts making their reuse in different context arduous. At the contrary, asynchrony promotes the re-use of components since the modules that compose a system can work independently and synchronize only when necessary through loosely coupled interfaces [4, 26]. These interfaces impose less constraints on their environment than their synchronous counterparts.

1.2 Synchrony versus Asynchrony

In reactive programming, the synchronous approach is justified by the fact that reaction times and interaction with the external environment must be precisely controlled. Components are thus tightly coupled with a global logical clock. In this model of computation we assume that a system interacts with its environment by performing global computation steps. In each step, the system reacts to environment stimuli by propagating their effects through its components complying with the causality order. The synchrony assumption stipulates that the system’s reaction is sufficiently fast with regards to the environment.
In practice, this means that events arising between step boundaries are processed at the next step and implies that responsiveness and precision are truncated by step duration. Verilog and VHDL and the so-called synchronous languages such as Esterel, Lustre and Signal follow the synchronous paradigm. These languages are well-suited for digital signal processing and automatic control applications. Synchronous programs are typically implemented as a single task that executes a read–compute–write loop. No real-time kernel is necessary since static scheduling is sufficient to resolve concurrency of components. For correct implementation, synchrony assumption should be verified by ensuring that the execution time does not exceed the clock period.

The asynchronous paradigm arose from the multitasking execution model. It does not impose any notion of global execution step. The concurrent processes are executed according to the pseudo-parallel model and communicate generally by message passing. Therefore, this paradigm is particularly suitable for distributed systems. Languages such as SDL, ADA and C adopt the asynchronous paradigm. SDL [68, 28] is particularly interesting since the communication statements are already part of the language. When a language does not support concurrency explicitly, these operators are provided through the use of thread libraries or through the use of a real time kernel.

It is advantageous to have a reference model written in SDL for two reasons:

1. Many standards like TETRA, UMTS, GSM, DECT, ISDN are described in terms of SDL.

2. SDL has formal semantics [45, 46, 34] and is therefore amenable to formal verification [31, 51, 43, 33] like model checking [11, 14].
1.3 Golden reference model

Traditional design starts from written specifications. These specifications have a number of well-known drawbacks. They are written in natural language and are by definition ambiguous. The model based approach plays a major role in removing ambiguity and incompleteness of the initial specification. The latter is gradually transformed into an executable model. The fact that the model is graphical [62] greatly improves the communication between team members during the validation of the user requirements. In fact, the earlier a problem can be identified, the easier and cheaper it is to fix. For this reason, the executable model is intensively simulated, tested and sometimes model checked [78] by system-level teams. The model becomes then a reference that enables a high level sign-off with implementation teams. The goal being to preserve the functionality of this golden model all over the design cycle. Another advantage of having a reference model, is to avoid architectural surprises later in the development phase. Moreover, an executable and testable reference model enables a precise reporting of progress throughout the product development process and a documentation that is always synchronized with the actual code as well.

1.4 Problem statement

As mentioned earlier, a significant effort is invested in the validation of a golden reference model. Therefore, one of the main challenges of the model-oriented approach is to preserve the correctness of such a reference throughout the design phases [38]. Embedded systems often require very small footprint that can be achieved only via manual coding. The latter is an error prone process. Automatic code generation does meet the performance constraints of many applications, but the compiler itself is not necessarily trustworthy. In effect, the user base of code generators used in model-based methodologies and domain-specific languages is necessarily small in comparison with the one of main stream compilers like gcc. The robustness of such code generators is henceforth
questionable. As a consequence, we are highly interested in ensuring that the integrity that has been achieved at the golden model is not compromised by a faulty manual or automatic translation. From the computability theory standpoint, program equivalence is undecidable for most interesting cases [81]. Nevertheless, one of the main challenges of this thesis is to seek for additional assumptions under which such equivalence checking can be algorithmically solved for industrial sized software projects.

1.5 Alternative solutions

Three approaches are possible for checking the adherence of the target code to its original specification namely testing [88], formal compiler verification, and finally translation validation [71, 70, 74, 64, 91].

1.5.1 Testing

By using the specification running in its simulation environment and the generated code running in its target platform then we may use back-to-back testing to increase our confidence in both the execution environment as well as the code generator. We extract test-cases from the specification, execute the test-cases on both the specification and the generated implementation, and compare the results. Any discrepancy between the executions indicates a problem with either the specification execution environment (i.e.; SDL simulator), the code generator or the implementation run-time environment. The associated flow is depicted in figure 1.4. The SDL language is often combined with the TTCN [73] language that is dedicated to build test-benches. A TTCN code generator is then required to build C test harnesses to verify the generated code. When the SDL tool-set does not provide an SDL simulator per se like the Telelogic Tau environment, the current method has absolutely no value. By simulating only the generated C, we will never be able to find any discrepancy between the actual SDL specification and the generated C code. Moreover, when the generated C code does not implement the SDL semantics many errors may go undetected. Consider for instance the following SDL code containing a conditional state-
ment. The problem is that the two alternatives are not mutually exclusive which should cause a dynamic error when simulating the SDL.

The C code generator has replaced the SDL decision by an if-else-if chain which is a perfectly legal C code. The second conflicting alternative will simply never get executed and no error will get reported at run-time.

```
decision m; 1
  [(=T1)] : 2
    output o1; 3
  [(=T2)] : 4
    output o2; 5
enddecision; 6
```

```
yVarP->yDcn_z020= yVarP->z021_m; 1
if ((yVarP->yDcn_z020) I == (•<) I) { 2
  yOutputSignal = xGetSignal((&ySigR_z2_ol),...); 3
  SDL_Output(yOutputSignal, (xIdNode *) 0); 4
}
else if ((yVarP->yDcn_z020) == (0) ) { 6
  yOutputSignal = xGetSignal((&ySigR_z2_o2),...); 7
  SDL_Output(yOutputSignal, (xIdNode *) 0); 8
}
```

Even with the availability of a simulator for the abstract model, the method presented in this section suffers from the well-known shortcomings of testing in general. In effect, test cases development is a frustrating and time consuming process. That is why testing remains by far the main bottleneck in all the developments in the industry. It is common for companies to dedicate around 50% of their overall resources for testing. Moreover, the resulting test-cases are necessarily redundant at times and incomplete at others especially when concurrency features are at stake. In safety-critical application’s domain, testing requirements may go further by imposing to certify the development tools like compilers and linkers as well. Tools should experience the same level of testing that is required for the application itself.

**Compiler certification**

Certification is a practical and proven mean of establishing a meaningful level of trust in the runtime behavior of software components. An emphasis on process certification illustrates the use of an indirect measure of software quality [55]. For safety critical systems such as avionics or automotive industry, special development standards and processes have been designed to minimize the dangers associated with those systems. All safety standards define a set of activities that have to be carried out in order to achieve a desired safety level.

Those activities can be generally grouped into three categories:

1. selecting development methods and tools
1.5. Alternative solutions

2. implementing the system

3. verifying and validating the system

For instance, DO-178B avionics standard imposes the same testing requirement for any tool that replaces a manual step in the development of the embedded software as it is imposed on the software itself. As a result, the output of such tools can be used without additional verification. However, qualifying a code generator involves an enormous workload which is reflected in very high cost of the project. For example, it was estimated that level A structural coverage testing of the Gnat compiler for Ada’95 consisting of about 625’000 lines of Ada and C code will require 58 person years of effort [76].

The price of a certified code generator is between five and ten times higher than that of a regular one [7]. Moreover, the qualification procedure for code generator is also very time consuming and maintenance cycle for such a generator is currently around two years while approximately three months are sufficient to produce corrected version for an uncertified code generator. The alternative way is to subject the application software to the prescribed verification activities as if no code generator had been used in its creation. The verification process used for this can also find errors that might be introduced by an automatic code generator.

1.5.2 Formal compiler verification

A compiler specification $S$ consists of formal descriptions of its source intermediate and target languages and the transformations from one language to the next. Languages must be specified by their operational [87] or denotational semantics [80]. This compiler speci-

![Figure 1.5: Compiler verification](image)

fication must be implemented correctly. We start first by verifying the compiler specification $S$ by showing that for every program $P$, every program $P'$ obtained from $P$ by $S$ is a correct translation of $P$. Secondly, we must prove that the compiler implementation $i(S)$ is a refinement of the compiler specification. Finally, it must be shown that the executable compiler is itself a correct translation of the compiler implementation.
We have to accept that up to now rigorous compiler verifications [23] have been performed mainly in academic institutions or in companies close to academic circles and not subject to hard industrial competition. There are virtually no executable industrial-commercial compilers available for which both specifications and implementations have been rigorously verified in full extension.

1.5.3 Translation validation

As pointed out earlier, formal verification of a compiler is not possible due to its size, frequency of change and also proprietary issues. As an alternative, translation validation appears to be a promising field in compiler verification. The approach was inspired by the run-time result checking technique [86] and more generally by the observation that it is often easier to check the output of a program than to verify the correctness of the program itself. Following this principle, we restrict ourselves to verifying that the output of a compiler is correct with respect to a given input source code. Verification occurs at each invocation of the compiler. The problem is that the input and output of a compiler are themselves programs with complex semantics and with different level of abstractions. A careful design of an intermediate representation coercing common aspects of the two programs is therefore required. Another requirement consists of relating states of both the input and the output programs at the source code level and propagating it at the intermediate level. Only at this stage, proof-obligations can be generated according to a given correct implementation relation. When proof-obligations fall into decidable theories the verification process can be fully automatic by using decision procedures. While translation validation does not prove that the compiler is correct, it still offers great engineering advantages compared to methods described in the last subsections.

1.6 Background

Compilers and code generators are complex piece of software that do contain errors. Bug databases of standard compilers [2] give an evidence of the previous claim. The GNU compiler collection announced that over 900 bugs have been fixed in release 3.4.0. This
1.7. Thesis contribution

includes all components in the collection including C, C++, Java and Ada compilers and also all their associated libraries [47]. Testing the compiler thoroughly implies a monumental effort. Let us consider for instance table 1.1 illustrating the number of test-cases needed to verify that a compiler is compliant to a given standard. Such test-suites reflect indirectly the effort involved in testing a real production compiler. We argue that code

<table>
<thead>
<tr>
<th>Language</th>
<th>Standard</th>
<th>Test-suite</th>
<th>Number of test-cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>ISO</td>
<td>CVSA</td>
<td>64’000</td>
</tr>
<tr>
<td>C++</td>
<td>ISO</td>
<td>C++VS</td>
<td>73’000</td>
</tr>
<tr>
<td>Java</td>
<td>Sun Microsystems</td>
<td>JETS</td>
<td>29’000</td>
</tr>
<tr>
<td>SDL’88</td>
<td>Z.100</td>
<td>Cinderella SDL</td>
<td>500</td>
</tr>
</tbody>
</table>

Table 1.1: Conformance to standards: Test-suites

generators for model-based languages including domain specific languages will never benefit from the same level of testing as regular main stream compilers which make them less robust.

1.7 Thesis contribution

Our research builds on previous and ongoing work on compiler verification and translation validation. But, rather than simply focusing on the synchronous or purely sequential programs, it addresses the parallel asynchronous composition aspects as well. The main contributions of this thesis are:

1. **Verification methodology** Developers can automatically verify the refinement of an abstract reference model into the final target code. By doing so, there is absolutely no need to re-validate again the target code obtained manually or by using a compiler. Without loss of generality, we treat the case where SDL is the source language and C the target implementation language. Both languages have gained wide acceptance in industry [79, 77, 90, 50, 84].

2. **Cutpoints-based-Equivalence checking** Their usage is a key concept in our method since they allow to subdivide the verification problem into smaller sub-problems. We use cutpoints to break the loops as in Floyd’s method but also to relate the concrete program to its abstract version.

3. **Working assumptions** In order to achieve full automation, we have imposed several simplifying assumptions. Those assumptions are commonly adopted in practice. The main assumptions that underly this research concern the restriction to a subset of SDL, the compilation scheme and the scheduling regime.

4. **Intermediate representation** To enable equivalence checking both abstract and concrete programs are translated into an intermediate form. This intermediate representation which we called APN coerces relevant aspects of both languages and is defined in terms of its syntax and operational semantics.
5. **Equivalence relation formalization** An inductive argument based on bisimulation has been formalized for the sequential case. Two cases of the asynchronous parallel composition have been also treated namely the server model and the activity thread model.

6. **SCEC Prototype tool implementation** To demonstrate the effectiveness of our approach a tool called SCEC has been developed. It is mainly written in C together with Flex and Bison specifications to produce the scanners and the parsers for C and SDL. A WxWidgets based graphical user interface has been developed in order to browse the intermediate representation of the programs. Starting from the syntax tree, SCEC can record all the transformations that are applied on the trees up to the final normal form. Proof-obligations are generated automatically and are proved using the ICS (Integrated Canonizer and Solver) tool. Currently SCEC handles the Telelogic CAdvanced compiler only. Rewrite rules are hard coded inside the tool. We claim that using a term rewriting engine [82], we can easily retarget other code generators.

### 1.8 Significance

We have identified two main scenarios where our approach could be used:

1. **Compiler testing** By introducing the rewrite-rules corresponding the compiling specification into SCEC, the testing effort will be reduced since it will be sufficient to feed both the compiler and SCEC with source test-cases and let SCEC check automatically the target code. The testing process may go further by using some heuristics to generate randomly and automatically source code test-cases for which SCEC will assess the correctness of the translation.

2. **End-users trustworthy compilation** Each Translation bug discovered by end-users will be fed back to the developers to repair the faulty translator module. This will improve the overall robustness of the compiler since the testing rate and test source code diversity will get increased.
This chapter provides a concise literature survey of concepts and views related to this thesis. Section 2.1 introduces the usage of equivalence checkers in hardware. In fact, equivalence checking prevails all formal verification techniques applied in hardware design. Due to the importance of the subject, a dedicated workshop called *Equivalence checking and assertion-based verification* is held every year to discuss the latest verification algorithms and their associated practical tools. We will therefore analyze in this chapter some techniques that allowed the scalability of equivalence checkers and what made them systematically used in current verification flows.

Section 2.2 addresses translation validation applied to the verification of the gcc optimizing compiler. This compiler allows to dump a machine independent assembly code after each invocation. The validation infrastructure verifies the compilation results for each enabled optimization and then asserts or negates their equivalence. In Section 2.3, we consider translation validation from the synchronous language *Signal* to its purely sequential version described in the C language. Finally, in section 2.4, we will see how *Bounded Model Checking* is used to verify the consistency between a behavioral circuit description in C and its associated *Verilog* version.

### 2.1 RTL Versus Netlist Equivalence Checking

Formal equivalence checking is a process, used commonly during the development of digital integrated circuits [8, 75, 16, 40], to formally prove that two representations of a circuit design are exchangeable without compromising the correctness of the global design. The circuit is described using a hardware description language like VHDL or Verilog at the Register-Transfer-Level.

RTL is a description of a circuit in terms of data flow between registers, which store information between clock cycles in a digital circuit. Once the RTL description has been verified by the logic designers by simulations and other verification methods, this reference design is transformed into a netlist by a logic synthesis tool. The initial netlist will usually undergo a number of optimizations. This netlist becomes the basis for the placement of the logic elements into a physical layout.
In theory, a logic synthesis tool guarantees that the netlist is logically equivalent to the RTL source code. In practice, programs have bugs and it would be a major bet to assume that all steps from RTL all the way to the final tape-out netlist have been performed without errors. Also, in real life, it is not uncommon for designers to make manual changes to a netlist, thereby introducing a major additional error factor. Instead of blindly assuming that no mistakes were made, a verification step is needed to check the logical equivalence of the final version of the netlist to the RTL source code.

In the past, the way to check the equivalence was to run the test cases that were developed for verifying the correctness of the RTL code on the netlist. This process is called gate-level simulation. However, the problem with this is that the quality of the check is only as good as the quality of the test cases. Also, gate-level simulations are slow to execute, which is a major problem as the size of digital designs continues to grow exponentially. An alternative way to solve this is to formally prove that the RTL code and the netlist synthesized from it are the same. This verification process is called formal equivalence checking. For combinational circuits, equivalence checking do scale with industrial-designs [37, 48, 15, 3, 41].

A major practical breakthrough came with the introduction of cutpoints. Given two combinational circuits, presumed to be structurally similar, whose functional equivalence needs to be verified. The idea is to seek for points in the two circuits which can be proven to be equivalent. The equivalent logic is cut out of the circuits and is replaced by a new primary input. If we can repeat this process all the way to the primary outputs, we have proven the two circuits equivalent, thereby reducing the original verification problem into a sequence of simpler verification problems.
In the next sub-section, we will address the equivalence checking method which treats the sequential case.

### 2.1.1 State machine equivalence

Two designs, modelled as finite state machines, are functionally equivalent if they exhibit identical output sequences for all valid input sequences. Checking this equivalence requires that the reachable states of both machines are computed jointly to prove the absence of state pairs with different outputs. State traversal is computationally expensive (PSPACE-complete) In practice, the state traversal becomes intractable when the number of state bits exceeds a few hundreds.

Two issues help reduce the algorithmic for large designs. First, in typical design methodologies, it is relatively inexpensive to require that the state encodings used in both machines be identical and requiring that the names of the state bits be preserved. In doing so, the general sequential checking can be reduced to combinatorial equivalence checking which is computationally less expensive. A second advantage of practical design methodologies is that the specification of one the machines typically includes a significant number of internal nets that have functionally equivalent counterparts in the description of the other machine. This similarity can be exploited for partitioning the combinational verification problem into smaller sub-problems.

### 2.1.2 Product machine

Let $M = (I, O, S, S_0, \delta, \lambda)$ denote a finite state machine where $I, O, S$ and $S_0$ represent the set of inputs, outputs, states and initial state respectively. $\delta : I \times S \rightarrow S$ denotes the next-state function and $\lambda : I \times S \rightarrow O$ is the output function. To check that two FSMs $M_1$ and $M_2$ with identical inputs and outputs are equivalent, we construct a product machine.
Chapter 2. Related work

\[ M = (I, O, S, S_0, \delta, \lambda) \]

in the following way:

\[
\begin{align*}
I &= I_1 = I_2 \\
S &= S_1 \times S_2 \\
S_0 &= S_{0,1} \times S_{0,2} \\
O &= \{0, 1\} \\
\delta(i, (s_1, s_2)) &= (\delta_1(i, s_1), \delta_2(i, s_2)) \\
\lambda(i, (s_1, s_2)) &= \begin{cases} 
1 & \text{if } \lambda_1(i, s_1) = \lambda_2(i, s_2) \\
0 & \text{otherwise.}
\end{cases}
\end{align*}
\]

\[ M_1 \text{ and } M_2 \text{ are said to be functionally equivalent, if and only if the output function of } M \]

produces one for all its reachable states and inputs. Verifying the equivalence is achieved by identifying a characteristic function \( g(s) \) that relates the states of \( M_1 \) and \( M_2 \). We have the following theorem:

**Theorem 1.** The product machine \( M \) produces \( \lambda \equiv 1 \) for all reachable states iff there exists a function \( g(s) : S \rightarrow \{0, 1\} \) such that:

\[
\begin{align*}
g(s_0) &= 1 \\
g(s) = 1 \Rightarrow \forall i. g(\delta(i, s)) &= 1 \\
g(s) = 1 \Rightarrow \forall i. \lambda(i, s) &= 1
\end{align*}
\]

When this theorem holds, the \( g \) provides an inductive invariant that proves equivalence of both machines. Different methods for equivalence checking differ in the way candidates for \( g \) are obtained. In practice, it is often the case to rely on the design methodology to infer \( g \). For instance, if retiming is applied as a synthesis transformation, the retiming function computed by the synthesis algorithm can serve directly as a candidate for \( g \) without the compromising the soundness of the approach. In general, it is common to require that both machines to be compared have identical state encoding and that there exists a direct correspondence between their registers.

2.1.3 Register correspondence

Although incomplete, register correspondence method allows to assess the equivalence of two state machines by performing simple combinational checks as illustrated in figure 2.3. Register correspondence establishes an equivalence relation on the set of registers of the product machine. Let \( REG \) be the set of registers of the product machine and let \( RC \subseteq REG \times REG \) be the relation that represents the register correspondence. A given \( RC \) provides the following candidate \( g = \land_{(s^i, s^j) \in RC} (s^i = s^j) \). In the absence of naming conventions, \( RC \) can be computed automatically as greatest fixed point using the following algorithm:
Algorithm RegisterCorrespondence {

$RC' = \{(s^i, s^j) | s^i_0 = s^j_0\}$;

do {

$RC = RC'$;

$\varphi = \bigwedge_{(s^i, s^j) \in RC} (s^i = s^j)$;

$RC' = \{(s^i, s^j) | (s^i, s^j) \in RC \land \forall x \forall s (\varphi(s) \rightarrow (\delta^i(x, s) = \delta^j(x, s)))\}$;

} while ($RC' \neq RC$);

return $RC$;

}

The algorithm starts with one equivalence class (registers whose content match initially). During each iteration, if the next-state functions of members of one equivalence class differ, this class is partitioned. This process is repeated until a fixed point is reached. Since the partitioning is monotonic and the number of registers of the product machine is finite, the algorithm is guaranteed to terminate.

2.2 Assembly code optimization translation validation

The translation validation infrastructure TVI presented in [65] checks an assembly code against its optimized version. The two assembly programs are obtained from a C program using the GNU C compiler (cf. figure 2.4). TVI is based on the intermediate representation whose abstract syntax is described in table 2.1. The state of the memory can be modified either by writing a value at a specified address through upd or by calling a function via updcall. The function sel denotes the dereferencing operator. The main requirement for checking the equivalence is the existence of a basic blocks correspondence.
(cf. figure 2.5). The latter relation associates basic blocks of related programs where each element of the relation is augmented with equality constraints between temporaries at both sides.

Two executions are equivalent if both lead to the same sequence of function calls and returns. Two returns are the same if the returned value and the state of the memory are the same. Two function calls are the same if the state of the memory prior to the call, the arguments and the address of the called function are the same in both cases.

The checker proceeds forward in parallel on both source and target programs. Synchronization is guided by the basic block correspondence. All associated constraints must be compatible with the global effect of each symbolically executed basic block. The verification ends when both programs hit a return instruction, in which case the state of memory and the returned value on both sides should coincide.

If all the constraints succeed then it can be shown by an inductive argument that the program fragments are equivalent modulo that given correspondence relation. The main difficulty lies in coming up with the correspondence relation. The most reliable way to do that is to have it produced by the compiler. Another alternative is to use some heuristics to decide on-the-fly whether a branch in the source program was either eliminated and
then which side was retained or if it was copied to the target program in the same form or in reversed form. The assumption being that all branches in the target program must correspond to branches in the source program. Those heuristics are based on conditionals and instruction sequence similarities. Similarity results are not straight booleans but rather scores between 0.0 and 1.0. Concerning conditional similarity, the conjunction *and* is implemented as multiplication and the disjunction *or* is implemented as the maximum operator. Given two instruction sequences:

- if the sequence of calls in one sequence is not a *prefix* of the related sequence then the lowest score is assigned.
- if both sequences lead to points that are already known to be related then the highest score is set.
- otherwise, each instruction is assigned a unique serial number. The similarity score is expressed in terms of serial numbers expressions of related sequences.

The checker defined in [65] was run on both the source code of gcc 2.91 and the linux kernel 2.2 which represents 5 million assembly code instructions. The resulting false alarm rate is claimed to be below 10%.

### 2.3 Signal to C translation validation

[72] proposes a translation validation framework called CVT for the synchronous language Signal[71, 89, 69]. CVT takes advantage from the fact that a Signal program has a restricted explicit control structure to provide a mechanical verification against the generated C program. The latter consists of a single main loop whose body is a loop free program. One loop iteration represents the effect of one synchronous step of the abstract program. The generated C code correctly implements the original Signal program if the

![Figure 2.5: Assembly code translation validation](image-url)
sequence of states obtained at a designated control location corresponds to a possible sequence of states in the abstract system.

**Signal to C translation**

A Signal program manipulates sequences of values called flows. A flow is always synchronized by a clock. More generally, any expression $\text{exp}$ has its corresponding clock $\text{clk}(\text{exp})$. A flow may be absent at some clock instant which is denoted by the symbol $\perp$. Clocks are special kind of flows assuming the values $T$ and $\perp$. A clock has a value $T$ iff the flow associated with the clock holds a value at the present instant of time. Signal has five primitives (cf. table 2.2) whose semantics will be presented in the next subsection. The compilation to C proceeds as follow. The statements of a signal program $P$ form a set of logical equations $SLE$ on the flows of $P$ and their associated clocks. Solution of $SLE$ for a given set of input register values determine the next state of the system. The compiler derives from $P$ a C program which consists of one main loop whose task is to repeatedly compute $SLE$ solutions. In order to do so, the compiler computes from the $SLE$ a conditional dependency graph on flows and another linear equation which records the dependencies among clocks. The order of assignment of variables representing the flows in the C program is henceforth consistent with the flow dependency graph. The control structure is completely determined by the clock of flows. The C code introduces explicit boolean variables to represent the clocks of Signal flows.

**Intermediate representation**

CVT translates both Signal and C programs into transition systems. A transition system $\langle V, \theta, \rho \rangle$ consists of a set of variables $V$ an initial condition $\theta$ and finally a transition relation $\rho(V, V')$. The set of variable $V$ is split further into $I,O,R,M$ and $L$ sets which denote the set of input, output, register, memorization and local variables respectively. Each signal primitive contributes to the transition relation according to table 2.3.

**Refinement relation**

The correct implementation relation is defined in terms of a refinement between two transition systems. Let $\mathcal{O}$ be an observation function mapping abstract and concrete states into a common data domain $\mathcal{D}$. An observation $\text{Obs}(A)$ is a sequence of $\mathcal{D}$ elements.
which can be obtained by applying the observation function \( O \) to each of the states in a computation of \( A \). Intuitively, \( C \) refines \( A \) if \( \text{Obs}(C) \) is a subset of \( \text{Obs}(A) \). In other words, every observation of \( C \) is also an observation of \( A \). CVT restricts the observation function to the IO sets and proposes to instrument the generated C code in order to detect the presence/absence of Signal flows (i.e., \( \bot \)). A pictorial representation of state correspondence is given in figure 2.6. The C transition relation is obtained by composing the transition relation of individual statements inside the loop.

![Figure 2.6: State correspondence](image)

**Table 2.3: Signal semantics**

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v := f(u_1, \ldots, u_n) )</td>
<td>( \text{clk}(u_1) = \ldots = \text{clk}(u_n) )</td>
</tr>
<tr>
<td>( w := v \oplus \text{init } u_0 )</td>
<td>( m.r' = f(\text{clk}(v')) ) then ( v' ) else ( m.r )</td>
</tr>
<tr>
<td>( v := u \text{ when } b )</td>
<td>( r' = f(\text{clk}(v')) ) then ( m.r ) else ( \bot )</td>
</tr>
<tr>
<td>( w := u \text{ default } v )</td>
<td>( v' = \text{if } b' = T \text{ then } u' \text{ else } v' )</td>
</tr>
<tr>
<td>( P\mid Q )</td>
<td>( [P] \land [Q] )</td>
</tr>
</tbody>
</table>

2.3.1 Verification conditions

The refinement relation can be expressed as an inductive argument (cf. table 2.3.1) provided the existence of an abstraction mapping \( \alpha \) relating concrete states to abstract states [49, 13]. CVT relies on the knowledge of the transformations applied by the compiler to infer the data abstraction mapping \( \alpha \). Under the previous assumption, the IOR variable sets can always be regenerated from the C program. The refinement is proven if a decision procedure can assert the validity of the following proof obligations:

\[
\theta_C \land \bigwedge_{r \in R} (m.r = r_c) \land \bigwedge_{v \in \text{IORL}} (v = \bot) \rightarrow \theta_A
\]

\[
\rho_c \land \bigwedge_{r \in R} \left( (m.r = r_c) \land (m.r' = r_c') \right) \land \bigwedge_{v \in \text{IO}} (v' = (\mathcal{O}_c^e)' \land \bigwedge_{v \in \text{RL}} (v' = eq_v) \rightarrow \rho_a
\]
\[ V_A = \alpha(V_C) \quad \text{Data abstraction} \]
\[ \theta_C \land V_A = \alpha(V_C) \rightarrow \theta_A \quad \text{Base case} \]
\[ \rho_C \land V_A = \alpha(V_C) \land V'_A = \alpha(V'_C) \rightarrow \rho_A \quad \text{Induction step} \]
\[ V_A = \alpha(V_C) \rightarrow \mathcal{O}^A = \mathcal{O}^C \quad \text{Compatible observations} \]
\[ C \sqsubseteq A \quad \text{Refinement relation} \]

**Table 2.4: Inductive argument for proving the refinement relation**

The boxed conjunctions refers to the fact that every abstract variable \( v \in RL \) has in principle a defining expression \( eq_v \) all in terms of the concrete variables \( V_C \).
2.4 Behavioral consistency of C and RTL Verilog

An I/O equivalence checker between a C behavioral description of a circuit and its Verilog RTL version is presented in [18]. The flow is depicted in figure 2.7. After transforming both programs into SSA form, the transition relations of both representations are unwound up to a certain depth $k$. The transformed relations are equations that represent the set of states that is reachable in exactly $k$ steps. Nested loops in the C program are first converted into a single loop by introducing new counters that keep track of which inner-loop should be executed. C pointers are recursively dereferenced using standard points-to analysis. No cycle accuracy between the two models is required. For the same input, outputs matching may require different execution steps on both sides. Resulting transition systems contain visible (i.e.; I/O actions) and hidden transitions as well. The proposed I/O equivalence is similar to weak bisimulation as described by Milner [60]. Using predicate abstraction [5, 22, 21, 20, 6], variables are abstracted to bit-vectors. Given an I/O mapping between the C and Verilog representations, the equivalence criterion is reduced to a safety property of the product machine of the two transition systems. To sum up, I/O equivalence checking was reduced to determining the validity of a bit vector equations thanks to Bounded Model Checking techniques [9, 17, 10]. Bounded Model Checking only shows the absence of inconsistencies up to a given bound. However, determining if this bound is large enough to guarantee the absence of any inconsistencies is not trivial.

**Conclusion** Translation validation applied to the synchronous language Signal cannot be generalized to other programming languages used to develop software. Actually, the synchronous model is identical to the zero-delay model of circuits[83] and is therefore very close to a hardware description. The verification of the optimizing compiler using translation validation approach cannot be generalized either since the semantic of two assembly programs (dumped from gcc) is identical. In addition, the fact that gcc itself provides the intermediate representation makes this approach very specific. In practice,
all the difficulty lies in designing the right intermediate representation that bridges two models with significant semantic gap. In effect, when the source and destination representations are very different, like in the case of behavioral C circuit description and its RTL verilog representation, then the correspondence between the two models is hard to find. Therefore, we fall into a model checking problem on the product machine of both circuits with a safety property stating the I/O equivalence that needs to be verified.

We believe that the work presented in this thesis is novel since it offers a practical solution to automatically verify the translation from the SDL language which contains concurrency through asynchronous message passing, to a scheduler-based C implementation. To the best of our knowledge, nobody solved this problem. The presence of queues between SDL processes together with the fact that state machines manipulate data with complex data types make the translation into process algebra oriented environment like CWB unfeasible. In our approach, we will use a cutpoint-oriented technique and additional hypotheses on the interleaving of the processes and their state encoding to reduce the computational complexity of the validation procedure.
This chapter presents the main features of SDL that are frequently used in practice. Since its definition in the seventies, SDL has gained a wide acceptance in the telecommunication standardization bodies and in the industry as well. SDL benefits from a sound semantics foundation and integrates itself well in the model driven architecture (i.e.; MDA) framework. In this framework, SDL is considered as a domain-specific language and is integrated as a UML profile. At the end of this chapter, concurrency aspects are addressed.

Concurrency is achieved through a runtime structure and through the usage also of an external scheduler. The SDL runtime environment includes data structures that are used by the scheduler to store and retrieve the execution context of the communicating state machines. This chapter is a pre-requisite for understanding the intermediate representation.

3.1 Overview

SDL is an object-oriented and formal language standardized by the International Telecommunications Union (ITU) as recommendation Z.100. The language is able to describe the structure, behavior and data of real-time and distributed communicating systems with a mathematical rigor that eliminates ambiguities and guarantees system integrity. SDL has been widely accepted specification technique for the software design of telecommunication systems and communication protocols in particular. The development of SDL started in the seventies and is extended in a four years cycle. SDL has two syntactical forms: the graphical representation SDL/GR and the textual representation SDL/PR. The graphical representation is more intuitive than the textual one and is therefore more widely used.

3.2 SDL abstraction levels

The strength of SDL is its ability to describe the structure, interfaces, behavior and data of a system.

3.2.1 Structure

SDL includes structuring mechanisms that enables the decomposition of a given design into sub-systems. An SDL specification can be structured in such a way that the decomposition reflects the actual structure of the system to be implemented. SDL distinguishes
three levels: system, block and process levels. The top-level description called system
denotes the frame of the description. This levels describes the global scope of the de-
sign and its interfaces to the environment. A system consists of a set of interconnected
blocks. The blocks in turn may contain sub-blocks thus forming a tree-like specification
hierarchy whose lowest level contains the communicating processes. Identifiers defined
in outer structures are visible in inner structures. An example of such SDL system is
given in figure 3.1. The system contains two blocks $B_1$ and $B_2$ connected using channels
$c_1, c_2$ and $c_3$. The additional pages are added for type and signal definitions. The block $B_1$
contains processes $p_1$ and $p_2$ that are connected using signal routes $r_1, r_2$ and $r_3$. In this

![Figure 3.1: Structure of an SDL system](image)

example, blocks and processes are statically wired. A more flexible solution consists in
defining block types and processes types whose instances can be interconnected through
their gates. New systems can be composed in this way by just changing the interconnec-
tions between sub-components.

### 3.2.2 Interfaces

Signals can be exchanged between blocks, processes and the environment following com-
munication paths. Those paths can be either channels or signal routes. Channels may de-
lay signals as opposed to signal routes. Blocks are connected through channels whereas processes within a block are connected using signal routes. The channels connected to a block are denoted by channel identifiers outside the block. There is always one or more signal routes connected to a channel. The signals on the routes to/from a channel must have identical names and direction of the signals on the channel. It follows therefore that all the signals on the routes to/from the block must be defined outside the block.

Signals usually have an attached list of information. The signal definition gives a name to the signal type and a list of predefined sorts of data or user defined sorts. In addition, every signal implicitly carries information about the identity of the source process. Signals which are output to a communication path are received in exactly the same order as they are sent. There may be several sources for a path, but each signal is placed into the path at the time of output and there can be no overtaking. Following the destination address determined at the time of output, signals are delivered to the receiver process at the end of the communication paths, and is placed in the input queue of that process. Whether it is received at this point depends on the content of the queue and the status of the process.

3.2.3 Data

All data in SDL are based on the concept of abstract data types (ADT). An ADT is basically a set of data values and associated operations that are precisely specified independently from any particular implementation. The ADT principle is therefore very well suited to a specification language. SDL also includes a number of predefined types. A few of these are specific to SDL while most are usual data types known from imperative programming languages. SDL is a strongly typed language. This means that conversions between different types are forbidden. There are no global variables in SDL. All variables are encapsulated in a process. Variables are updated exclusively by signal consumptions.

3.2.4 Behavior

The lowest level of the SDL specification hierarchy consists of one or more processes. A process is a finite state machine extended with locally defined and typed data variables. To each process instance, an input queue is associated implicitly. This input queue stores incoming signals and timer signals (i.e.; timeouts). The behavior of an SDL specification is completely determined by the set of process instances that are interconnected according to the channels and signal routes discussed previously. Processes are disjoint and run concurrently. They interact with other processes by exchanging messages, called signals. Figure 3.2-a shows symbols commonly found in an SDL process.

Example of such process is depicted in figure 3.2-b. $P_1$ interprets the action after the start symbol and then cyclically moves from one state to another. Every process must have one and only one start. When a process is initiated, the variables for the process are initialized, then, any actions in the transition between the start state and the first state are interpreted. When a process is in a state, each signal in the input queue is examined...
in turn. For each state, the process diagram defines how to handle each type of signal. Unless some signals are being saved, then the first signal is consumed and the transition corresponding to the input containing that signal is interpreted leading to a next state. A task is used to manipulate data, typically evaluating expressions over local data. A decision branches according to an expression in the decision, taking the alternative which corresponds to the value of the expression. An output directs a signal to another process. A stop terminates the process.

Unlike a system or block diagram, a process diagram is typically spread over several pages and connectors are used in classical flowchart style to join from one page to another. The statement join has the same effect as a goto in a programming language, namely to direct the flow of control to a point that is marked by a label, the name of which is given in the join construct. This kind of jump is allowed only within a process body or a procedure. A label name is thus local for a body and must be distinct, of course from other label names in the body. Label and join are mainly used for layout purposes. To obtain a printable format of the specification, SDL allows the spreading of the graphical representation of a big state machine into several pages. On the other hand, factorization is also possible through some short-hand notations. This is explained in the following section.

Figure 3.2: (a) Process symbols (b) Example of SDL process
3.3 Layout mechanisms

3.3.1 State multi-appearance

On a process diagram the state may appear more than once in a state symbol, provided that no two occurrences are followed by the same input signal. Usually this will be done if there are too many inputs to show on a single page. In this case the extra inputs are shown on additional pages. Another reason for multiple appearance of a state is so that all the states followed by a particular input can be shown on the same page.

3.3.2 State shorthand notations

When the same transitions follow the same signal in different states, then all the states may be listed in a single state symbol. This provides simplification of the diagram, particularly when a large number of states are followed by the same transition for a specific signal. This will almost always be used with multiple appearance of the states, and can be used to produce a diagram where each page is relevant to a particular signal rather than a particular state. If the same transition follows all states then an asterisk is used in the state symbol. Similarly there is a notation for all states except a specified list of signals. Finally, in the next state symbol, a dash means that the state does not change.

3.3.3 Input shorthand notations

Similar to shorthands for states, there are shorthands for inputs. When the same transition follows different inputs there can all be put in the same input symbol. If the same transition follows all explicit signals except those already mentioned in other inputs and saves then this can be denoted by an asterisk. In the following section, we will discuss how processes communicate together.

3.4 Communication mechanisms

3.4.1 Outputs

Outputs are used to send signals to other processes. The signals sent may influence the behavior of the receiving process. The output may have actual parameters carrying typed values. Theses values carried in the constructed signal will assign values to variables of the receiving process. A signal that is sent is expected to be received by some other process instance. Versions of SDL until SDL-88 required that it was just one process instance that was finally the recipient of the signal, because otherwise it would result in a dynamic error. Form SDL-92 onwards, signals that do not find a destination process instance are discarded. If the signal cannot find a unique process for delivery, an arbitrary process out of the possible ones is selected. There are two basic kinds of addressing:

- explicit addressing, in which the address of a particular process instance is uniquely specified. The ascertains delivery except for the case when the process instance ceases to exist before the signal reaches its destination.
• implicit addressing, in which the address of the receiving process is not uniquely specified. Variants of this addressing may however restrict the possible receiving processes by prescribing routes for the signals or by naming a process set that receives the signal.

Signals that do not find a receiving process instance are discarded and signals that find more than one of the possible receiving process instance will choose one of the possible receivers in an arbitrary, non deterministic fashion.

### 3.4.2 Queuing signals handling

Consider the process diagram represented in figure 3.3. Signals A, B and C placed in the input queue. A state $S_1$, a signal of type A is expected. The signal $A$ in the queue is therefore consumed immediately. The queue is then empty, but $B$ and $C$ arrive while the transition to $S_2$ is being interpreted. At state $S_2$, the input queue is examined and the first signal found $B$ is not saved and therefore consumed. Signal $C$ remains in the input queue and when the state $S_3$ is entered, the input queue is again examined and signal $C$ is consumed. In general, a signal that is not expected in a given state is discarded (i.e.; implicit consumption). Since SDL processes are running asynchronously, signal may get lost. To cope with this problem, SDL provides a save construct.

### 3.4.3 Save

The `save(sig_type)` construct allows the input queue to retain signal instances of type `sig_type` in the queue. This violates the FIFO principle but allows to postpone the processing of certain signals. The save construct has the same shorthands as the input construct, but of course both saves and inputs can be used on the same state so there are additional rules which apply to their use in combination. For each state in the set of input symbols and save symbols:

![Figure 3.3: Queuing signal handling](image)
3.4. Communication mechanisms

1. each signal can be mentioned only once.

2. asterisks can only be used in at most one save or one input.

A useful construct is an asterisk state followed by an asterisk save. This has the effect of saving all signals not explicitly mentioned for any state. In that case, signals are not implicitly consumed and can therefore be considered as persistent. The save construct usage is exemplified in figure 3.4. At state $S_2$, signal $C$ is consumed. The signal $B$ is saved at the front of the input queue. Since $B$ is not expected at state $S_3$ then $B$ is discarded according to the implicit transition principle.

![Figure 3.4: Queuing with Save](image)

3.4.4 Timers

A timer provides a mechanism for inserting a signal into the process input queue at some future time. When a timer is set to a value of sort time then the system clock is continuously monitored until the timer expiration. At expiry, a signal with the same name as the timer is inserted in the input queue. Normally, a timer will be set to a number of time units from the current value, now. This is done by adding a duration value to the now pre-defined operator using a set action. A reset action stops the timer. If the timer signal is already in the input queue, the reset action removes the timer signal from the queue.

3.4.5 Enabling condition

The enabling condition construct makes it possible for the user of SDL to impose an additional condition for the initiation of a transition. A transition is initiated by an arriving signal only if the boolean expression has the value true. Otherwise the signal is saved and the process remains in the same state (cf. figure 3.5). Values carried by a signal cannot be used in the enabling condition.
3.4.6 Continuous signal

So far, a transition is initiated by an arriving signal. However, there are situations where the user of SDL may wish to show that a transition is initiated when a certain condition is fulfilled instead. This can be done by using a continuous signal. A continuous signal contains a boolean expression preceded by the keyword provided. A transition is initiated when the input queue is empty or when all signals are saved and the boolean expression has the value true. A priority clause may be added when more than one continuous signal is defined for the same state. Figure 3.6 gives the intuition behind it. As long as the condition $c$ is not fulfilled, an implicit signal $cs$ is continuously sent to the local input queue.

3.4.7 Import and export

In SDL, a variable is visible only within the process in which it has been declared. If another process wants to access the value of the variable, a signal exchange with the process owning the variable is needed. By using the import/export shorthand, the user of SDL can specify this in a much simpler way.
This concludes the list of SDL features that we often encounter in practice. In the next section, the structure of the SDL runtime will be discussed.

SDL is not only used for modeling and simulating systems, but SDL models can also be turned into a software implementation. Since SDL is a concurrent language, a scheduler is needed on the target platform to emulate the pseudo-parallel execution of the SDL processes. Therefore, the SDL runtime structure describes the data structures that will serve as interface between the SDL processes and the scheduler.

3.5 Runtime structure

The runtime structure depends on the SDL constructs that are needed for implementing the system under consideration. The more general runtime environment which is used in simulation follows the server model. For more constrained environment like embedded systems, the activity thread model is often used.

3.5.1 Server model

In the server implementation model, each SDL process is implemented as a single software Real-time operating system (RTOS) task. The term server model is chosen for this straightforward implementation scenario because here each SDL process acts like a server, which exclusively waits for requests in the form of triggering SDL signals.

For auto-generated C code from SDL, an adaptation layer between the C functions describing process activities and the actual kernel is often necessary. Commercial kernels provide messages queues working in first-in first-out mode only. The save construct requires an additional RTOS queue that needs to be polled before the main input queue. In addition, timers are handled centrally by a dedicated task.

The Telelogic CAdvanced runtime environment implements the server model. Figure 3.7 depicts the corresponding process descriptor structure. The runtime environment maintains a list of such process descriptors. Each process descriptor refers to a process type whose instances denoted Inst manipulate disjoint sets of data (i.e., local data of the process). Each instance has a separate field which stores its explicit state cs.

The extended finite state machine of the SDL process is implemented as a combination of a switch-case statement over a transition identifier together with a set of look-tables. The main table contains the set of states. Each state refers to the following elements:

- the transition table denotes the map associating a signal identifier to its transition number TrId in case of normal input (NI).
• the transition type table associates each signal identifier to the transition type. Four values are possible: save, normal input, unexpected signal or enabling condition (EC).

• the EC evaluator field is present when the transition type refers to an enabling condition (EC). This field refers to a function that evaluates the condition under which a signal may be consumed. This function returns also the transition identifier in case the condition evaluates to true otherwise the signal is saved.

• the CS evaluator field is present when a continuous signal is used. This field points to a function that evaluates the condition expressed when using a continuous signal. This function returns the transition identifier in case the condition is fulfilled otherwise the transition is not fireable.

When a signal reaches the head of the queue, its processing is performed in two phases. First, the transition identifier is resolved using the look-tables and then the actual processing is handed over a behavior function which will modify the local data of the process instance and will determine also the successor state.

3.5.2 Activity thread model

The activity thread technique implements an SDL specification as a set of functions where for each input signal a corresponding function is defined. The active elements in this model are the signals. An incoming signal activates the corresponding function which immediately handles the signal and when producing an output calls the respective function of
the next entity. The sequence of inputs and outputs results in a sequence of function calls
called activity thread. Note that the term activity thread does not refer to an operating sys-
tem thread. It denotes the execution path a signal triggers in the set of processes. A pure
synchronous activity thread technique provides very efficient implementations because it
avoids the storing of signals when calling the next entity but requires the reordering the
actions within the transitions like it is done in [56].

 Basically, all output statements are shift after the state change operation to avoid sig-
nal losses and signal overtaking. By using an activity thread scheduler, we can get an
asynchronous implementation by using a common signal queue that is filled at each out-
put statement. Basically, the scheduler looks up the front of the queue and calls then the
appropriate function in case it is ready to consume the signal.

The Telelogic CMicro runtime environment implements the activity thread model. Fig-
ure 3.8 illustrates the associated process descriptor. The look-tables are more compact. Only signals that are expected in a given state are represented. A special value of TrId in-
dicates whether or not a signal needs to be saved. There is neither support for the enabling
condition nor for the continuous signal.

![](image)

**Figure 3.8: Process descriptor used by the CMicro code generator**

**Conclusion** SDL is a language that promotes re-use by making each SDL process com-
pletely independent from its surrounding processes. Reusing a process in a different con-
text requires only to provide identical signal routes at its interfaces. The concurrency
model is simple and easy to analyze. Inter-process communication is always built upon a
signal exchange. No process can modify local data of other processes without triggering a
signal exchange. Once a process has consumed a signal, other processes cannot interfere
on that process to change the successor state. We will see in the following chapters that
these interesting features form a sound basis for defining a precise notion of equivalence.
On the other hand, the analysis of the SDL runtime environments shows that SDL fits nicely the MDA framework regarding the platform independent model (PIM) and platform specific model (PSM). There is a clear separation of concerns between the specification phase and the implementation phase.
This chapter presents an important contribution of this thesis which is the design of the intermediate representation. We had to make the choice between either defining a set of low-level statements upon which we can build higher level constructs of both SDL and C or unifying the constructs of both languages. Due to the significant semantic gap between SDL and C, the first solution would have made the correspondence between the two descriptions very hard. Therefore, we have chosen the second alternative of merging the constructs of the source and the target language. Since SDL supports concurrency, we described the intermediate representation using a process algebra style. Three operators are presented namely the prefix, the choice and the parallel composition operators. In our intermediate representation, a process represents an extended finite state machine. When these processes are coupled together using the parallel composition operator they form what we call an Asynchronous Process Network (i.e.; APN)

4.1 Asynchronous Process Network

An Asynchronous Process Network is a set of processes that communicate with each other and with the environment asynchronously using signals and queues. Two scheduling modes are considered: the server model and the activity thread model. In the server model, we associate a single queue per process whereas in the activity thread model one global queue is shared by all processes. The communication is asynchronous in the sense that the sending process is not blocked by its output action. Sender and receiver are neither synchronized by a global clock nor there is a rendezvous to exchange data.

An APN contains one or more processes. Each process is an extended finite state machine that is driven by the content of its associated queue. When no data is present in the queue, predicates on local data may be defined as transition triggers. The control flow of an APN is defined using three operators. First, the parallel composition operator $\psi$ represents the scheduler that controls the global execution of processes. Second, the choice operator $+$ represents an alternative between two transitions inside a process and finally, the prefix operator denoted $\cdot$ captures the notion of sequential execution within
4.2 APN syntax

\[
\begin{align*}
PN &::= P \mid \psi (P \_list) \\
\text{P}\_\text{list} &::= P \mid P\_\text{list} P \\
P &::= \text{Choice}\_\text{list} \text{ Save}\_\text{list} \\
\text{Choice}\_\text{list} &::= \text{P}\_\text{Term} \mid \text{P}\_\text{Term} ^+ \mid \text{Choice}\_\text{list} \\
\text{Save}\_\text{list} &::= \text{save}(i) \mid \text{save}(i) \text{ Save}\_\text{list} \\
\text{P}\_\text{Term} &::= C \mid T . C \\
T &::= \text{input} \mid \text{guarded}\_\text{input} \mid \text{background}\_\text{transition} \\
\text{input} &::= i \cdot (\text{'signal}\_\text{params}) \cdot [R] \cdot [\text{Guard}] \cdot i \cdot [R] \\
\text{guarded}\_\text{input} &::= [] \cdot \text{Guard} \cdot [] \cdot \text{input} \cdot [R] \\
\text{background}\_\text{transition} &::= [] \cdot \text{Guard} \cdot [R] \\
R &::= \text{Guard} \\
\text{Guard} &::= -i \cdot b\_\text{term} \mid b\_\text{term} \text{ LogOp b}\_\text{term} \\
C &::= \text{C}\_\text{Term} \cdot C \mid \text{nextstate} P \mid \text{join} \text{ label}\_\text{term} \\
\text{b}\_\text{term} &::= \text{C}\_\text{Term} \mid \text{C}\_\text{Term} \text{ RelOp} \text{ C}\_\text{Term} \\
\text{C}\_\text{Term} &::= v \mid f \cdot (\text{C}\_\text{Term}\_\text{list}) \\
\text{C}\_\text{Term}\_\text{list} &::= \text{C}\_\text{Term} \mid \text{C}\_\text{Term} \text{ C}\_\text{Term}\_\text{list} \\
\text{LogOp} &::= \lor \mid \land \\
\text{RelOp} &::= > \mid < \mid \leq \mid \geq \mid = \mid \neq \\
f &::= \text{assign} \mid \text{fset} \mid \text{set} \mid \text{reset} \mid \text{add} \mid \text{sub} \\
&\mid \text{output} \mid \text{label} \ldots
\end{align*}
\]

An APN includes a set of processes that are combined using the parallel composition operator denoted \(\psi\) (cf. line 1). A process is represented by a set of mutually recursive state definitions. One state is elected to be the initial state. At each state, there is a choice (i.e.; +) between several transitions. A list of saved signals may also be specified on a per state basis. A transition can be split into a trigger \(T\) followed by a sequence of computations \(C\) (cf. line 5). The trigger (cf. line 6) is either a normal input optionally guarded or a condition on local variables.

For the same input signal we may have more than one successor state because SDL allows the presence of conditional statements under an input signal. For this reason we have introduced a reachability predicate to attribute a trigger. This predicate determines uniquely the path to the future state among all possible paths. In other words, a transition trigger can be viewed as a macro-transition composed of several micro-transitions (cf. section 4.2.2). A sequence of computations \(C\) is always closed with either a nextstate statement or a join statement (cf. line 12). A join statement refers to a connection or to a control edge end point.
4.2. APN syntax

4.2.1 Prefix and Choice operators

Table 4.1 illustrates prefix and choice operators using SDL symbols. For example, \( P = \bar{a} \cdot Q \) means that process term \( P \) may receive signal \( a \) then becomes term \( Q \). A process may have a choice between several alternatives. For example the following process expression \( P = a \cdot Q + b \cdot R \) means that process \( P \) may either receive \( a \) then moves to \( Q \) or may receive \( b \) then moves to \( R \). Axioms for prefix and choice operators are defined in table 4.2.1. The choice operator is commutative (A1) and associative (A2). Prefix operator is associative (A3) and distributes over the choice operator from right (A4).

\[
\begin{align*}
\text{Prefix: } & P = a \cdot Q \\
\text{Choice: } & P = P_1 + P_2
\end{align*}
\]

\[\begin{array}{c}
p \\
\times \\
\circ \\
\bullet \\
q \\
\end{array}\]

\[\begin{array}{c}
\land \\
\circ \\
\bullet \\
\circ \\
\bullet \\
\end{array}\]

Table 4.1: Choice and prefix operators

4.2.2 Reachability predicate

Figure 4.1 exemplifies the notion of reachability predicate. The SDL process contains three macro-transitions under input \( i \), input \( j \) and continuous signal \( k \) respectively. Macro-transition under input \( j \) contains six micro-transitions. For instance, the micro-transition included under input \( j \) representing the path from state \( S_0 \) to state \( S_2 \) is determined by the reachability predicate \( R_{02} = \neg b \land \neg a \). The internal representation has been built in such a way that each state in the abstract model contains a set micro-transitions that need to be matched with the corresponding set of micro-transitions of the concrete state. Another example depicted in figure 4.2 including communication, decision and computations shows how the reachability predicate is derived by performing backward substitution starting from the upmost guard up to the input statement. In effect, the condition \( x > 0 \) becomes \( x > -2 \) after substituting \( x \) for \( x + 2 \) due to the assignment statement. In case of nested decisions, we must start by lifting first the upmost guard then proceed downwards. Each condition inside an SDL decision will contribute with one conjunct to the global reachability predicate.

\[
\begin{align*}
A1 & \quad P + Q = Q + P \\
A2 & \quad (P + Q) + R = P + (Q + R) \\
A3 & \quad (P \cdot Q) \cdot Z = P \cdot (Q \cdot Z) \\
A4 & \quad (P + Q) \cdot Z = P \cdot Z + Q \cdot Z
\end{align*}
\]

Table 4.2: Axioms of the prefix and choice operators
4.3 APN Semantics

4.3.1 Operational semantics

Conventions

- $S$: State set
- $I_x$: Signals that can be processed at state $x$
- $S_x$: Signal saved through save construct at state $x$
- $E_x$: Signal saved dynamically through guarded input construct at state $x$
- $H_x$: Signals saved at state $x$
  - $H_x = S_x \cup E_x$
- $D_x$: Signals discarded at state $x$
- $\mathcal{I}$: input set of a process
  - $\mathcal{I} = I_x \cup H_x \cup D_x$ holds for all $x$ in $S$
- $\mathcal{I}^*$: Input set closure
  - $\mathcal{I}^* = \mathcal{I} \cup \mathcal{I}^2 \cup \mathcal{I}^3 \cup \ldots$
- $\lhd$: Concatenation function over signal sequences
  - $\lhd: \mathcal{I}^* \times \mathcal{I}^* \rightarrow \mathcal{I}^*$
- $Q_P$: Input queue associated to process $P$
- $Q$: Tail of queue $Q$
4.3. APN Semantics

We use operational semantics to assign a meaning to an APN. The abstract state vector includes local data \( ld \) and the communication state \( Q \) represented by all signals in transit (i.e.; waiting at the input queue of the process). In figure 4.3.1, we present a template that will be used in following sub-sections. Each statement is preceded by pre-conditions at control location \( \pi_0 \) and followed by post-conditions at control location \( \pi_1 \) (i.e.; Hoare triple). Parallel blocks represent the effect of the choice operator.

**4.3.2 Input/Output semantics**

In Table 4.3, we present the semantics of input and output statements. The input statement \( i(p)[\mathcal{R}] \) refers to a signal of type \( i \) carrying a number of typed parameters \( p \). As mentioned in section 4.2.2, the signal \( i \) encapsulates a set of micro-transitions. Each micro-transition under signal \( i \) is fully determined by its associated reachability predicate \( \mathcal{R} \). At \( \pi_0 \), we require that the front of the queue is a signal instance of type \( i \) and that the reachability predicate related to \( i \) evaluates to true. After the execution of the input statement, the signal instance \( i \) is removed from the head of the queue and the actual signal parameters replace local data variables that are referred to in the input statement (i.e.; \( \bar{p}/d \)). To define the semantics of the output statement, we need to refer to the state of the receiving process. We assume that the receiver process can be obtained at compile time.

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S )</td>
<td>( S )</td>
</tr>
<tr>
<td>( \pi_0 )</td>
<td>( \pi_0 )</td>
</tr>
<tr>
<td>( \mathcal{C} )</td>
<td>( \pi_0 )</td>
</tr>
<tr>
<td>( i(p)[\mathcal{R}] )</td>
<td>( \pi_1 )</td>
</tr>
<tr>
<td>( \pi_1 )</td>
<td>( \pi_1 )</td>
</tr>
<tr>
<td>{ ( \pi_0(Q) = Q \cap (i(p)) ) \</td>
<td>{ ( \pi_0(Dest(i)) = P ) \</td>
</tr>
<tr>
<td>( \pi_0(ld) = d )</td>
<td>( \pi_0(Q_p) = Q_0 )</td>
</tr>
<tr>
<td>( \mathcal{R}(d) )</td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>{ ( \pi_1(Q_p) = (i(\bar{p})) \cap Q_0 )</td>
</tr>
</tbody>
</table>

*Table 4.3: Input and output semantics*
by applying the oracle function \( \text{Dest} \) to the signal type passed as argument to the output statement. The effect of an output consists of appending a signal instance at the back of the queue of the receiving process. Signals that are not expected in the current state are discarded (cf. Figure 1.4). Two cases need to be distinguished. First, when the queue contains a sequence of signals built from only \( D_A \) and \( H_A \) then the new sequence is obtained from the original sequence by deleting all elements from \( D_A \). Second, when the queue contains at least one signal from \( I_A \), then the queue is partitioned by the closest element \( i \) to its front that can trigger a state change. The first case applies then to the front sequence \( j \).

### 4.3.3 Background transition semantics

This captures the semantics of the SDL continuous signal statement. When the input queue is empty or when all the signal instances are saved (cf. figure 4.5) in the current state then a condition on the local data can trigger a transition. This feature is interesting since it allows to define a background behavior of a process in absence of stimuli coming either from the surrounding processes or from the environment.

### 4.3.4 Guarded input semantics

Table 4.4 represents the semantics of guarded input which mimics the SDL enabling condition statement. When the condition \( g \) associated to the input evaluates to true (cf. table 4.4 case a), then we are in the case of normal input consumption defined in section 4.3.2.
4.3. APN Semantics

The signal \( i \) is otherwise saved in the current state. This saving is dynamic and depends on \( g \) and should not be mixed up with the effect of the save construct which is purely static. To handle the insertion and the deletion of the dynamically saved signal into/from \( E_X \) set, we need to refer to the transition that is fired in the current state. Fire-ability is expressed using the predicate \( F \) whose semantics can be inferred from the previous sections.

### Table 4.4: Guarded input semantics

<table>
<thead>
<tr>
<th>Guarded input (a)</th>
<th>Guarded input (b)</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Guarded input (a) diagram" /></td>
<td><img src="image2" alt="Guarded input (b) diagram" /></td>
</tr>
</tbody>
</table>

\[
\text{Guarded input (a)}: \quad \{ \pi_0(\varnothing) = \varnothing \cap \{ (i(p), ð) \} \}
\]

\[
\text{Guarded input (b)}: \quad \{ \pi_0(\varnothing) = \varnothing \cap \{ (i(p), ð) \} \}
\]

### 4.3.5 Parallel composition operator semantics

Let \( \mathcal{P} \) be the list of processes to be composed by the parallel composition operator \( \psi \). The \( \psi \) operator can be viewed as a recursive function that moves \( \mathcal{P} \) from one configuration to another. A configuration is defined by the content of the queues and the local data variables associated to individual processes. For each configuration, we define \( V_k \subseteq 2^{\mathcal{P}} \) as being the set of fireable processes. The function \( \psi \) uses internally a choice function \( \sigma \) that picks up randomly one element from \( V' \), fires the selected process and then moves to the next configuration. The following rule defines the semantics of the parallel operator \( \psi \):

\[
\psi(\pi_0, \ldots, \pi_k, \ldots, \pi_m) \rightarrow \psi' (\pi_0, \ldots, \pi_k', \ldots, \pi_m)
\]

This inference rule for \( \psi \) is the most generic description of a scheduler following the server model where each process has unique input queue. In practice, we tend to restrict the non-determinism resulting from the parallel composition by imposing some extra rules like process prioritization or even the Early Message Deadline First [54].

In a classical implementation, it is common to use the Activity Thread model that removes completely the non-determinism by sharing one single global queue between all the processes. Figure 4.6 illustrates the difference between the server model and the activity thread model. Three processes \( A, B \) and \( C \) are represented. For each process we have defined the corresponding I/O map. For example, \( a \rightarrow fx \) means that that process \( A \) can dequeue the signal \( a \) and then outputs the signal \( f \) to process \( B \) and then outputs...
the signal $x$ to process $C$. We initiate the execution by inserting the signal $a$ into the two instance set of these three processes. The first set of instances denoted $\mathcal{A}$ runs under the server model whereas the other one which is denoted $\mathcal{C}$ runs under the activity thread model. The traces of both $\mathcal{A}$ and $\mathcal{C}$ are represented for three derivations of the parallel composition operator of both systems. After the first derivation, we have $\mathcal{P}_A^1 = \{B, C\}$ and $\mathcal{P}_C^1 = \{B\}$. These two sets become $\mathcal{P}_A^2 = \{A, B, C\}$ and $\mathcal{P}_C^2 = \{A\}$ respectively after the third derivations. We remark that the first element of the shared input queue of $\mathcal{C}$ resolves completely the non-determinism. In the following chapters, this operator will be denoted by the symbol $\omega$.

**Conclusion** In this chapter, we have defined the syntax and semantics of our intermediate representation. The syntax uses concepts from process algebra whereas the semantics is defined using structured operational semantics. A process is built hierarchically using the *choice* and *prefix* operators following a top-down approach. A state within
a process is defined by an equation whose terms represent alternatives (choice operator) available when interacting with peer processes. Those alternatives are basically the micro-transitions leading to successor states. Each micro-transition is uniquely defined by its reachability predicate. Micro-transitions under the same trigger are further grouped into a macro-transition. Actions within one micro-transition are bound with the prefix operator enforcing the notion of sequentiality of execution within one transition. In other words, a process can be viewed as mutually recursive set of state equations.

The APN is built from the individual processes using the parallel composition operator in a bottom-up manner. This operator assumes that only one process gets executed at a time and relies on a choice function which selects one process to be executed among the set of fireable processes. It is common in practice to use the activity thread model for the implementation side. In this model, the scheduling is efficient since all the non-determinism is completely removed.
This chapter defines a bottom-up compositional verification method of the equivalence between two programs represented in APN form. For the lowest level, abstract and concrete processes are checked pairwise using an inductive argument. At the highest level where processes are executed concurrently, we present two specific cases where the interleaving between processes does preserve the invariant defined at the lowest level. We will review first Floyd’s method for verifying sequential programs and then draw the parallel with our equivalence checking problem. As in Floyd’s method, we use the notion of cutpoint to cut the loops but also to relate the concrete program to its abstract version. Cutpoints are at the heart of our method since they allow its scalability.

5.1 Cutpoints

Using cutpoints, the overall task is partitioned into a set of smaller verification problems which can be solved independently. Floyd’s inductive assertions method uses cutpoints augmented with assertions to prove the partial correctness of imperative programs. Finding those invariant is hard and cannot be automated. However, in our equivalence checking problem, we exploit the development methodology to derive using simple name matching the correspondence relation between cutpoints of both the abstract and the concrete programs.

For example, it is not expensive to require that the SDL state names and input names are preserved when transforming an SDL specification into a C program. Another requirement is to define an observation function $O$ that projects the program state into a relevant sub-set. Assertions are obtained now by equating the observation functions evaluated at related cutpoints. If the assertions hold for all related cutpoints then the equivalence proof is done up to the selected observation function. One drawback of such approach is that we exclude potentially equivalent processes for which we can not find a cutpoints correspondence.

5.1.1 Inductive assertions revisited

Theorem 2 (Inductive-Assertion method (Floyd)). For a given program $P$ whose input and output variables are $\bar{x}$ and $\bar{z}$, an input predicate $\phi(\bar{x})$, and an output predicate
\( \psi(x, z) \), apply the following steps: (1) Cut the loops (2) find an appropriate set of inductive assertions; and (3) construct the verification conditions. If all the verification conditions are true, then \( P \) is partially correct wrt \( \phi \) and \( \psi \).

A path between two assertions \( I_0 \) and \( I_1 \) is the set of statements executed when the program runs on all possible real data, such that the path starts with assertion \( I_0 \), ends with assertion \( I_1 \), and contains no other assertions. Since each loop is cut by an assertion, and since there are only a finite number of program statements, there are only a finite number \( N \) of such paths. Those paths correspond to a compile-time traversal of the program rather than a run-time traversal. The goal is achieved as follows: for each of the \( N \) paths, show that if assertion \( I_0 \) holds, and if the statements on the path are executed, then assertion \( I_1 \) holds. Given these \( N \) proofs, a straightforward induction argument on the number of paths traversed shows once and for all that these \( N \) proofs suffice to demonstrate that each assertion is true. This induction argument uses the transitivity of implication. Figure

5.1.1 illustrates the Floyd’s method on an SDL process implementing integer division. By sending the signal \( \text{div}(x, y) \) (\( x \) is the dividend and \( y \) is the divisor), the process sends back the signal \( \text{Result}(q, r) \) where \( q \) is the quotient and \( r \) is the rest of the division. We have four cutpoints \( I_n \). At these cutpoints, invariants have been formulated. We have three paths. The program is partially correct if the verification conditions are valid.

### 5.1.2 Observation function

The most obvious observation function \( \mathcal{O} \) that we could consider to prove equivalence is a function that focuses only on input and output actions (i.e.; I/O observational equivalence). In such case, two processes are considered equivalent if they exhibit the same I/O patterns. If we consider an external observer that interacts with the process as a black

---

**Figure 5.1: Computational induction**
5.1. Cutpoints

box then the input/output abstraction leads to non-determinism as depicted in figure 5.1.2.
The IR of the abstract process comes from the SDL process represented in figure 4.2. The assignment \( x := x + 2 \) is represented by the silent action \( \tau \). For simplicity, we consider a similar process representing the associated concrete process. For the same injected in-
put signal \( a \), we have different successor states. The successor state cannot be uniquely determined by the content of the input signal. Decisions inside a transition are based on computations performed on local data. We conclude that we cannot find a correspondence invariant in the case of I/O observational equivalence.

5.1.3 Setting-up the invariant

As explained in section 5.1, we move from the problem of finding invariants for each process in isolation to establishing an invariant that ensures a correct correspondence between paired processes. This is depicted in figure 5.1.3. We suppose that initially (i.e.; \( \alpha_0 \) and \( \beta_0 \)) local data of both processes denoted \( V_A \) and \( V_C \) are related by a data
abstraction mapping \( \iota : V_C \rightarrow V_A \). The initial cutpoints \( \alpha_0 \) and \( \beta_0 \) are the only points where communication with the environment can occur. This hypothesis is fundamental to our inductive argument. When both processes receive the same stimulus from the environment, they will perform some computations and then meet again at \( \alpha_1 \) and \( \beta_1 \). At this point we need to verify that our data mapping \( \iota \) still hold and that both processes have produced the same sequence of signals to their respective environment. If we iterate the procedure for all possible paired paths, then we can conclude that both processes are equivalent.

### 5.1.4 Cutpoints classes

Cutpoints are introduced at three levels (cf. figure 5.4). In level 1, \( A_i \) corresponds to an SDL process which is typically implemented in a C function \( C_i \). The other levels are exemplified in figure 5.5. States \( S_0 \) and \( S_1 \) together with the out-connector \( l_1 \) are level 2 cutpoints. The input \( i \) represents a macro-transition and belongs therefore to level 3. This SDL program fragment can be written using APN syntax as follows: 

\[
S_0 = t_0 + t_1 + t_2 \\
t_0 = i[a \land \neg b] \cdot \text{join}(l_0) \\
t_1 = i[a \land \neg b] \cdot \text{join}(l_1) \\
t_2 = i[a \land b] \cdot \text{nextstate}(S_2)
\]

We assume name matching for those three levels in the sense that the names are preserved when transforming the abstract specification into a concrete implementation. When such correspondence can be established, all what remains to do is to relate micro-transitions of the concrete and abstract APN processes together. Note that the second level includes also connection names. In fact, a connection is an SDL concept that allows to split the graphical representation of a process in a number of pages using in and out connectors. Using a connection name as cutpoint allows to reduce drastically the number of micro-transitions to be matched as depicted in figure 5.6.

If \( x \) is the number of paths occurring before the connection cutpoint and if \( y \) is the number of paths occurring after that cutpoint then we need to check \( x + y \) instead of \( xy \).
5.2 Process-level equivalence

Let \( T \) be the set of micro-transitions, \( S \) be the set of states, \( C \) be the set of connections and \( \mathcal{L} \) the set of labels. We define the micro-transition source function \( \alpha : T \rightarrow S \cup C \) and we define also the micro-transition destination function \( \beta : T \rightarrow S \cup C \cup \mathcal{L} \).

**Definition 1 (Micro-transition equivalence).** Two micro transition \( t_1 \) and \( t_2 \) are equivalent written \( t_1 \equiv t_2 \) iff:

1. \( \alpha(t_1) \) is related to \( \alpha(t_2) \).
2. \( \beta(t_1) \) is related to \( \beta(t_2) \).
3. the data abstraction mapping is preserved between their respective sources and destinations.
4. \( t_1 \) and \( t_2 \) produce the same sequence of signals to their respective environment.

**Proposition 1 (State Transition Graph).** Any process \( \langle s_0, S, C, \mathcal{L}, T \rangle \) can be transformed into state transition graph \( \langle S \cup C \cup \mathcal{L}, s_0, \rightarrow \rangle \) where the transition relation \( \rightarrow \) is defined as follows: \( x \xrightarrow{\ell} y \) iff \( \exists t \in T : \alpha(t) = x \land \beta(t) = y \).
Definition 2 (Simulation modulo $\preceq$). Let $G_{sdl} = (S_{sdl}, s_{0}^{sdl}, \rightarrow)$ and $G_c = (S_c, s_{0}^{c}, \rightarrow)$ two state transition graph. $G_c$ simulates $G_{sdl}$ if it exists a binary relation $\sim \subseteq S_{sdl} \times S_c$ such as:

- $\forall s_{sdl} \in S_{sdl}, \exists s_c \in S_c : s_{sdl} \sim s_c$
- $s_{sdl} \sim s_c \land s_{sdl} \xrightarrow{t_{sdl}} s'_{sdl} \Rightarrow \exists s'_c \in S_c : s_c \xrightarrow{t_c} s'_c \land s'_{sdl} \sim s'_c \land t_{sdl} \cong t_c$

if $G_{sdl}$ simulates $G_c$ via $\sim^{-1}$ then $\sim$ is a bisimulation.

In the following will use the symbol $\approx$ for bisimulation and the symbol $\sim$ for simulation.

5.3 Parallel-level equivalence

One intrinsic complication in analyzing concurrent programs is that when composing two processes to be run in parallel, we cannot derive input output relations computed by the composed program from just the input output relation computed by each of the individual component processes [58]. The reason is that the combined processes may interfere on each other altering the behavior each single process would have when run separately. The worst situation turns up when processes communicate through shared variables.

Moreover, communication in APN occur exclusively at the beginning of the micro-transition. At this communication point in particular, we require a strict equality of the inputs between related micro-transitions. If these two associated microtransitions are equivalent in the sense of definition 2, then, at their respective ends, we assert that they will modify the internal data of their embedding processes in the same way and that they will contribute with the same output signal instances to the external receiver processes.

5.3.1 Server model

When the server model is used for both the abstract and the concrete processes (cf. figure 5.7) then the following holds:

$$(\forall i \in \mathbb{N}_m^s : A_i \approx C_i) \rightarrow (\forall \psi \in \Psi : \psi(A_1, \ldots, A_m) \approx \psi(C_1, \ldots, C_m))$$

$\Psi$ represents the set of all possible schedulers. All computation terms are executed atomically. When a transition is selected the prefix operator ensures commitment (i.e.; run to completion) [25]. Atomicity and commitment are the two main conditions to ensure compositionality [24].
5.3. Parallel-level equivalence

Figure 5.7: Parallel composition operator: server model $\psi$

5.3.2 Server model versus Activity thread model

Figure 5.8 represents the case where the abstract processes use the server model whereas the concrete processes use the activity thread model. The latter uses the $\omega$ scheduler which is described in section 4.3.5. In this case, the abstract model simulates the concrete model but the converse is not true. In other words, we have a trace refinement between the abstract and the concrete model that is all the traces of the concrete model are included in the trace of the abstract model. The following implication holds:

$$(\forall i \in \mathbb{N}^m : A_i \approx C_i) \to (\exists \psi \in \Psi : \omega(C_1, \ldots, C_m) \sim \psi(A_1, \ldots, A_m))$$

which means that if the individual processes from the abstract and the concrete sides are pairwise bisimilar then there exists an abstract scheduler which uses the server model so that the parallel processes simulate the $\omega$ scheduled processes on the concrete side.

Conclusion We presented in this chapter a formalization of the notion of equivalence of two programs described in term of our intermediate representation APN. At the process level, we consider that two processes are equivalent if there exists a bisimulation up to $\cong$ that relates their respective states. In APN, states are connected through micro-transitions and not through abstract action names. Now, two micro-transitions with related source and destination states are considered equivalent ($\cong$) if they preserve the data abstraction mapping linking concrete variables to their abstract versions and also if they both produce the same output signal sequences to their respective environment.
The concept of cutpoints has been used to relate each abstract process to its concrete form via name matching. Cutpoints reflect the following top-down hierarchy: process, state/connection and finally transition trigger. Cutpoints are also used to break and relate the loops of both processes.

As far as concurrency is concerned, our bottom-up compositionality argument holds provided the two following conditions are fulfilled:

1. Processes do not interfere on each other outside communication points (i.e.; input statements)

2. Scheduling is done at a transition-level: Once the fireable transition is selected the process is run-to-completion.

**Figure 5.8:** Server model $\psi$ versus Acticity thread model $\omega$
This chapter describes the compilation process of both the SDL and the C programs into the APN form. Compilation is defined as a rewrite system. A rewrite system enables modularity since only local incremental transformations are performed on the syntax tree. Many structural transformations are required since the original SDL and C grammars are different. Translating SDL into the APN form is straightforward, since by construction APN was built in such way, that it subsumes a low level representation of SDL models. The emphasis in this chapter is on the C part, since the translation to APN is less obvious.

6.1 Compilation as a rewrite system

The compiler to APN form is structured as a set of very small passes, each of which performs a small part of the compilation process (i.e.; rewrite rule application). This method yields a more readable and maintainable compiler. Bugs that arise are more easily isolated to a particular task. Writing individual passes is easier since new code need not be grafted onto existing passes nor wedged between two logical passes that would have been combined in a monolithic structure.

One drawback of this approach is that the compiler is much slower than a typical production compiler since each pass requires a full traversal of the syntax tree. In our case, modularity is more important than performance from the robustness standpoint.

Formally, the compiler to APN can be viewed as a rewrite system which is basically a set of rewrite rules of the form $\alpha \rightarrow \beta$ where $\alpha$ and $\beta$ are input and output tree patterns respectively. When a sub-tree matches the input pattern $\alpha$, it gets transformed according to the tree pattern $\beta$. Trees and patterns are represented in the prefix linear form. For example, $f(x, y)$ denotes the tree with root $f$ and sub-trees $x$ and $y$. If $e$ is tree then $C(e)$ represents its context. Substituting sub-tree $\beta$ for sub-tree $\alpha$ in the context $C(e)$ is written $[\beta/\alpha]C(e)$. The rewrite rule implementation has the following form:
function a2β rewrite_rule (expr_t * e)  
{  
  if (e)  
  {  
    if (e->expr_tag == isfunc)  
    {  
      if (matches(e, a))  
        [β/ε]C(e);  
        a2β rewrite_rule (e->vof.fune.operands);  
    }  
    a2β rewrite_rule (e->next);  
  }  
}

We assume in the following sub-sections, that the symbol + occurring inside the rewrite rules denotes the Kleene repetition operator.

Example 1: Structured variable flattening  In SDL, variable declarations are local to the process. The Telelogic CAdvanced compiler moves all the declarations into a structured process descriptor (i.e., xPrsNode) to expose them to the scheduler. In order to establish the variable correspondence between SDL and C, we need a rule that transforms each field selection operator into a flat variable. Assume that x is a structured variable and that y is a field within that structure. fsel is a field selection operator.

$$\text{fsel}(x, y) \sim x \bowtie y$$

A fresh variable is formed by concatenating the two terms passed as parameters to the fsel operator. Concrete application of the rule is depicted in table 6.1. The implementation of the structured variable flattening rule is given in figure 6.1.

Example 2: Deref/Addr elimination  In the generated code from Telelogic CAdvanced compiler, SDL assignments of local data variables of the form $a := a_0$ are translated into expressions like $*(\&y\text{VarP} \rightarrow a) = a_0$ where $y\text{VarP}$ is a pointer to the structure representing the process descriptor manipulated by the scheduler. Such C expressions can be simplified using the following rule:

$$\text{Deref}(\text{Addr}(x)) \sim x$$
function fsel_yVarP_rewrite (expr_t * e) {
if (e) {
    if (e->expr_tag == isfunc)
        if (strcmp (e->name, "fsel") == 0)
            expr_t *fsel_arg0 = get_first_op (e);
            if (strcmp (fsel_arg0->name, "yVarP") == 0)
                expr_t *field = fsel_arg0->next;
                char *fresh_name = concat(field->narae, "yVarP");
                expr_t *sr_scope = e->Embeddingfunc;
                search_and_replace (e, sr_scope,NULL, new_ident (fresh_name), NULL);
    fsel_yVarP_rewrite (e->vof.fune.operands);
} fsel_yVarP_rewrite (e->next);
}

Figure 6.1: Structured variable flattening implementation

The symbol Deref is a function that returns the value pointed to by a its argument. Conversely, the symbol Addr returns a pointer to the location specified by its argument. The composition of these two functions is the identity function. Concrete application of this rule is depicted in table 6.2.

\[
\begin{array}{c|c}
\text{Deref} & 0 \\
\text{Addr} & \text{fsel} \\
yVarP & \text{z0014.cf} \\
\end{array}
\] 

\[
\begin{array}{c|c}
\text{Deref} & 0 \\
\text{Addr} & \text{fsel} \\
yVarP & \text{z0014.cf} \\
\end{array}
\]

Table 6.2: Deref/Addr elimination

Example 3: signal output reconstruction Sending a signal to another process in SDL is done using the output statement. This statement specifies the signal to be sent with all its actual arguments. In the generated C code, explicit signal instance construction is performed by a sequence of assignments that specifies the signal type and the setting of the signal instance’s arguments from the local data according to the signal signature. In
the following rule := (x, y) represents the assignment operator in prefix form:

\[ C := (fsel(sig, type), A), (:= (fsel(sig, a_i)))+, SDL.Output(sig, 0) \sim C(output(A(a_i))) \]

Concrete application of this rule is shown in table 6.3.

<table>
<thead>
<tr>
<th>fsel</th>
<th>ySigR.z4.MaUnitData.indication</th>
<th>fsel</th>
<th>yOutputSignal</th>
<th>type</th>
<th>Param1</th>
<th>yVarP</th>
<th>z013.sa</th>
</tr>
</thead>
<tbody>
<tr>
<td>yOutputSignal</td>
<td></td>
<td>yOutputSignal</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.3: signal output reconstruction

### 6.1.1 Compiling C FSM code to APN

Recall that each process/state in APN can be written as follows: \( P = \sum T \cdot C \) where \( T \) is the macro-transition trigger. \( C \) represents the computations performed when the transition is fired. \( C \) may lead to different successor states. Each successor state determines completely a micro-transition. In order to extract all micro-transitions, a traversal of the control flow graph (i.e.; CFG) associated to \( C \) is required. The latter problem will be explained in sections 6.2 and 6.3.

Let us consider the SDL program fragment represented in figure 6.2(a). We are interested in compiling the C code implementing the state machine \( P_1 \) into the APN form: \( S_0 = sig_0 \cdot C_1 + [B_0] \cdot C_2 + [B_1]sig_2 \cdot C_3. \) In this case a pure syntactic rewrite rule is not sufficient anymore. A precise semantic analysis is needed to compile C into the APN form. Therefore, knowledge about the internal encoding of the SDL specification by the compiler is crucial. At the implementation side, the SDL design is flat (i.e.; set of processes). The main FSM of the SDL process is implemented as a C function called yPAD (i.e.; Process Allocation Descriptor). The pseudo-parallel execution of the yPAD functions is handled by a scheduler. This scheduler fires one transition at a time by calling the selected yPAD. A transition is run-to-completion before selecting the next one. The context of execution is stored in a process descriptors list. Each element of this list is of type xPrsNode.
6.2. Control flow graph construction

Actually, the scheduler selects a transition by analyzing jointly the content of the input queues and the xPrsNode elements. A pictorial representation of P1’s xPrsNode is represented in figure 6.2(b). The first level includes the list of all input signals denoted xInputSignals and the state structure xStateIdStruct. Each state element contains a map indicating the trigger type per input signal (i.e.; yStaH). The interpretation of the integer values in yStaH is given in table 6.4 (a). For example, at state $S_0$, the signal sig0 is a normal input, sig1 is saved, sig2 is not expected and finally sig3 represents an enabling condition. The second component in the state structure is the map relating the trigger to the transition number (i.e.; yStal). For instance, transition number 1 refers to the trigger sig0 at state $S_0$ whereas transition number 4 refers to the trigger sig2 at state $S_1$. Figure

![Figure 6.2: (a) SDL process P1 (b) Process descriptor’s C data structure](image-url)

6.3 depicts the C implementation of P1. The conditions B0 and B1 are evaluated in the function yCont_S0 and yEnab_S0 respectively. Pointer to these two functions are stored in the xPrsNode and can be called by the scheduler independently from the yPAD function. The reason for this code motion optimization is to avoid calling a yPAD in a transition for which the firing condition does not hold. Based on the previous semantic analysis, the rewrite rule that is applied to the yPAD can be written:

$$\text{switch}(t, (\text{case}(t_i, \text{code}))) + \rightsquigarrow (P_m = \sum T_n \cdot \text{code}_i) +$$

where the tuple $(P_m, T_n, t_i)$ represent the state-event transition matrix. For the process P1, this matrix is represented in table 6.4(b).

6.2 Control flow graph construction

6.2.1 Threading the AST

The control flow graph can be constructed statically by threading the tree[36], as follows. A threading function exists for each node type. The threading function for a node of type $T$ gets a pointer to the node to be processed as a parameter, and calls the threading
functions of its children, in a recursive traversal of the AST. The set of functions maintains a global variable `last_node_ptr`, which points to the last node processed on the control flow path, the *dynamically* last node. When a new node $T$ on the control path is met during the recursive traversal its address is stored in `last_node_ptr.successor` and `last_node_ptr` is made to point to $T$. For example, the threading function for a binary expression has the following form:

```c
function Thread_bin_exp (Expr * Expr_ptr)
{
    Thread_expr (Expr_ptr^left_op);
    Thread_expr (Expr_ptr^right_op);
    /* Link this node to the dynamically last node*/
    Last_node_ptr -> succ = Expr_ptr;
    /* Make this node to the new dynamically last node*/
    last_node_ptr = Expr_ptr;
}
```

This makes the present node the successor of the last node of the right operand and then registers it as the next dynamically last node. Figure 6.4 shows the AST and the control flow graph for the expression $b^2 - 4ac$, where the pointers that make up the AST are shown.
as solid lines and the control flow graph using arrows. Note that the first node visited by the
threading function is the node *minus* at the top but that the first node in the control
flow graph is the leftmost *b*, as it should be.

A complication arises if the flow of control exists in more than one place from the
tree below a node. This happens, for example, with the *if* statement. Actually, there are
two problems here. The first is that the node that corresponds to the runtime then/else
decision has more than one successor. The second problem is that when we reach the
node dynamically following the entire *if* statement, its address must be recorded in the
dynamically last node of both the then-part and the else-part. So a single last node pointer
is no longer sufficient. The first problem can be solved by just storing two successor
pointers in the if-node. This makes the if-node different from other nodes. One way to
solve the second problem is to replace the last node pointer by a set of last nodes, each of
which will be filled in when the dynamically next node in the control flow is found. But it
is more convenient to construct a special *join* node to merge the diverging flow of control.
The threading function for an *if* statement (cf. figure 6.5) has the following form:
function Thread_if_statement (node * if_node_ptr) {
    Thread_Expr (if_node_ptr->condition);
    Last_node_ptr->succ=if_node_ptr;
    end_if_join = new (node);
    aux_last_node_ptr = last_node_ptr;
    Thread_Expr((if_node_ptr^then_part));
    if_node_ptr^true_succ = aux_last_node_ptr^succ;
    last_node_ptr^succ = end_if_join;
    last_node_ptr = aux_last_node_ptr;
    Thread_Expr((if_node_ptr^else_part));
    if_node_ptr^false_succ = aux_last_node_ptr^succ;
    last_node_ptr^succ = end_if_join;
    last_node_ptr = end_if_join;
}

The if-node passed as a parameter has two successor pointers: true successor and false successor. Note that these differ from the then-part and else-part pointers. The part pointers point to the tops of the corresponding syntax subtrees whereas the successor pointers point to the dynamically first nodes in these subtrees. The code starts by threading the expression which is the condition in the if-statement. Next, the if-node itself is linked in as the dynamically next node, last_node_ptr having being set by thread_expr to point to the dynamically last node in the expression. To prepare the processing of the then and else parts, an end_if node is built, to be used to combine the control flows from both branches of the if-statement and to serve as a link to the node that dynamically follows the if-statement. Since the if-node does not have a single successor field, it cannot be used as a last node, so we use a local auxiliary node aux_last_node to catch the pointers to the dynamically first nodes in the then and else parts. The call Thread_Expr((if_node_ptr->else_part)) will set the pointer to its dynamically first node in aux_last_node, from where it is picked up and assigned to if_node_ptr->true_succ by the next statement. Finally, the end of the then-part will have the end-if-join node set as its successor.

6.3 Micro-transitions generation

Given the control flow graph (i.e.; CFG) of an SDL process or a C function, we can derive all micro-transitions (i.e.; control flow paths) by a depth first traversal of the CFG. A stack is used to store the micro-transitions during the traversal. Micro-transitions under the same macro-transition share the same source which is the transition trigger and is closed by a nextstate or a join statement. For each macro-transition, the following algorithm is used to generate all micro-transitions:
function MicroTransGen (Node current, Stack s , PathContainer pc) {  
    push(s,current);  
    if (is_join_or_nextstate(current)) {  
        StorePath (pc,s);  
    }  
    foreach (current.successors) {  
        if (current.succ is not in s) {  
            MicroTransGen (current.succ,s,pc);  
        }  
    }  
    pop(s);  
}

In general, control flow path grows exponentially to the number of predicates in conditional statements. The fact that SDL processes are developed graphically helps subdividing the micro-transition matching problem into smaller sub-problems. In effect, a graphical representation of an SDL process is often split in several pages using in and out connectors. Micro-transitions matching is performed on a per page basis using the connection cutpoint concept. In practice, the number of micro-transitions within one page remains manageable.

6.4 Equivalence condition generation

In APN form, concrete and abstract programs are reduced to two sets of micro-transitions that have to be matched. Pairing between elements of the two sets is performed following the state/macro-transition hierarchy described in section 5.1. First, states/connections are matched using name matching. Second, inside each state/connection, embedded macro-transitions are related in the same way. Finally, within each macro-transition, micro-transitions are related pair-wise. Two micro-transitions are said to be equivalent if their joint logical formula is valid. The validity of such formula is discharged using a third-party decision procedure.

The verification algorithm implemented in SCEC traverses both micro-transition forward in parallel. Computational terms are matched syntactically. When guards are hit on both sides, they are fed into the decision procedure (i.e; on-the-fly) to decide on their equivalence. When the end of the two candidate micro-transitions is reached with successful syntactic matching and equivalent guards, the two paths are marked as equivalent. Henceforth, the matching paths are removed from the micro-transitions set to be matched. The emptiness of the latter set denotes the end of the checking procedure. Otherwise, micro-transitions that could not be matched indicate the presence of a discrepancy between the abstract and the concrete programs. This method works correctly only when the compiler does not change the order of instructions. The Telelogic CAdvanced compiler fulfill this requirement.
Example: Micro-transition matching Consider the following two micro-transitions that are related according to the data mapping $i = \{(a, x), (y, b), (z, c)\}$

\[
\begin{align*}
  a &:= b + c; & x &:= y + z; & 1 \\
  \text{guard} (a = 0); & & \text{guard}(x \geq 0 \land x = 0); & 2 \\
  b &:= b - a; & y &:= y - x; & 3 \\
  c &:= c + 1; & z &:= z + 1; & 4 \\
  c &:= 2 \ast b; & z &:= 2 \ast y; & 5
\end{align*}
\]

The algorithm takes first both terms $a = b + c$ and $x = y + z$. After performing substitutions according to the data mapping $i$, the term $a = b + c$ matches syntactically the term $[a/x, b/y, c/z](x = y + z)$. Therefore the two terms are equivalent. The next two terms are guards. These guards come from a conditional statement in the original source code and denote the chosen alternative. Although syntactically different they are still stated as equivalent by the decision procedure $(a = 0 \iff [a/x](x \geq 0 \land x = 0))$. The algorithm continues in the same way up to the end of the two micro-transitions. Note that, if we swap line 3 and line 4, then this algorithm will not be able anymore to assert the equivalence.

To handle data flow and control flow dependencies [63, 44], the two program fragments need to be translated into an assignment-free form where each use of variables is given a unique name. Each use of a variable to be reached by only one definition of that variable. This is very similar to the Static Single Assignment (i.e.; SSA) form used in compiler optimization. However, in our case the micro-transitions are straight-line code. There is no need to compute dominators to derive \(\Phi\) nodes. The equivalence formula that handle both control and data dependencies can be obtained by renaming the variables and propagating definition names to their uses.

\[
\left( (a_0 = x_0) \land (b_0 = y_0) \land (c_0 = z_0) \right) \quad \iff \\
\left( \begin{array}{l}
  x_1 = y_0 + z_0 \\
  x_2 = 0 \\
  y_1 = y_0 - x_2 \\
  z_1 = z_0 + 1 \\
  z_2 = 2 \ast y_1 \\
  (a_2 = x_2) \land (b_1 = y_1) \land (c_2 = z_2)
\end{array} \right)
\]

The left-hand side contains the conjunct $(a_0 = x_0) \land (b_0 = y_0) \land (c_0 = z_0)$ which asserts the data mapping between the two program fragments at the beginning of the micro-transitions. This mapping needs to be preserved at the end of both paths. This is represented by the conjunct $(a_2 = x_2) \land (b_1 = y_1) \land (c_2 = z_2)$. In other words, the data mapping has to be compatible by the two sequence of computations performed on both paths.

Conclusion We have shown in this chapter that the compilation into the intermediate representation can be expressed in terms of rewrite rules. Those rewrite rules can be deduced from the transformations performed by the compiler under verification. In this work, the rewrite rules have been hard-coded into the prototype tool and are specific to the
6.4. **Equivalence condition generation**

Telelogic CAdvanced code generator, but we believe that it is advantageous to integrate a term rewriting engine to retarget other compilers. In such situation, the user will provide the tool with rewrite rules related to its compiler and then let the rewrite engine perform the actual transformations on the abstract syntax trees.

In a second phase, micro-transitions are generated by a recursive traversal of the control flow graph. Finally, comes the micro-transitions matching phase. Pairing two micro-transitions is confirmed when the decision procedure (ICS) asserts their equivalence. Verification conditions belong to two decidable logics: logic of uninterpreted functions and linear arithmetic logic.
This chapter presents first the SCEC tool and its associated flow and then the results of two case studies: IEEE 802.11 MAC layer (terminal side) and the MIPv6 protocol. These two case studies were chosen to demonstrate that the selected SDL sub-set (defined in annex A) is sufficient to describe real-world protocols. During the following experiments, the generated C source code was verified against its SDL specification using SCEC. One major outcome is that during the verification process of the MAC layer, a subtle bug in the CAdvanced SDL2C compiler was discovered. This result is very encouraging and shows that the development of such equivalence checker does pay off.

7.1 SCEC Flow

Figure 7.1 depicts the flow used to verify the adherence of the C implementation with respects to its original SDL specification. First, SCEC generates the Abstract Syntax Tree for both SDL and C programs and performs standard semantic analysis. For the C part, AST comprises data type definitions, global data declarations, and complete function

![Figure 7.1: Flow](image-url)
bodies, whereas for the SDL part, SCEC stores type definitions, signals and process bodies. The ASTs are gradually transformed using rewrite rules. Some of them are generic while others are specific to the CAdvanced code generator. At the end of the rewrite process, we obtain on the one hand, a number of state transition graphs representing the SDL processes and on the other hand, the corresponding C functions (i.e.; yPADs) that implement them. All SCEC has to do, is to compare the SDL processes and the C functions pairwise. For each related process and function in APN forms, states, connections and macro-transitions are related in the same way. Finally, micro-transition are matched together if the proof obligation associated to both paths can be discharged by ICS.

7.2 Experiments

7.2.1 The WIFI MAC station case study

Figure 7.2 illustrates the upper-level SDL system design of the Wifi MAC layer within the Telelogic Tau environment. The lowest-level comprises 15 SDL processes. The MAC layer is standardized by the IEEE. The original SDL diagrams came from annex B of 802.11 specification [1]. The SDL textual representation (i.e.; PR form) represent about 4’000 lines of SDL code and are translated into around 17’000 lines of C code using the SDL2C CAdvanced 3.5 compiler form Telelogic.

Figure 7.2: 802.11 IEEE MAC for a station
SCEC timing performance Using SCEC, the whole equivalence checking process takes about 52.8 seconds on an Intel Centrino 1.5 GHz, since most of the verification conditions turn out to be trivial after relating the proper cutpoints. Figure 7.3 shows statistics extracted from the intermediate representation. We can see clearly that the number of micro-transitions that need to be matched by SCEC does not exceed a hundred of paths for the largest process (i.e.; TX coordination). Without the connection cutpoint explained in section 5.1.4, the verification takes about 422.4 seconds which represents an average slow-down factor of 8 compared to the optimized version. When connection cutpoints optimization is enabled, the gain depends heavily on the occurrence of the cutpoint in the code. For instance, if the cutpoint $c$ is between two consecutive conditional statements whose number of paths is significant, then $c$ plays a major role in subdividing the matching procedure into two independent parts. Concerning the MAC case study, most of this gain comes from the two processes Prepare_MPDU and Filter_MPDU for which the speed-up factors are 4 and 25.2 respectively. Detailed timing performance can be found in figure 7.4.

![Figure 7.3: macro and micro-transitions to be matched](image)

7.2.2 The MIPv6 protocol case study

MIPv6 is defined by IETF (Internet Engineering Task Force) in the Request For Comments RFC 3775. This protocol enables a node to remain reachable while moving from its Home network to other networks called Visited Networks. MIPv6 protocol contains three main components: The mobile node (MN) that represents any node able to move through the Internet network while maintaining its communications. The home agent (HA) is an access router of the home network that manages the accessibility at any time to the MN independently of its localization. The Correspondent Node (CN) represents any node that communicates with the MN. Figure 7.5 shows the SDL system of MIPv6
protocol. The SDL system contains 2462 lines of code and was developed by the authors of [67]. SCEC took about 48 seconds (on the same machine as before) to check the adherence of the generated C code against the SDL code. Results are shown in figure 7.6.

7.3 Finding bugs in the compiler

Thanks to SCEC we were able to find a bug in the Telelogic CAdvanced 3.6 SDL-to-C compiler during the validation of the case study presented in section 7.2.1. The bug was cross-checked against the Cinderella tool-set which exhibited a correct translation and execution. The simplest form of the bug is presented in the following SDL code. The type $T$ is equivalent to the boolean set with the definition of two literal synonyms of predefined values. A variable $t$ instance of $T$ gets assigned the value $X$. The conditional statement provides two alternatives in which the integer variable $s$ gets a different value.
7.3. Finding bugs in the compiler

newtype T inherits Boolean
  literals X = false, Z = true;
operators all;
endnewtype T;
dcl
  s integer,t T := X;
start;
  task t := X;
  decision t;
    (X) :
      task s := 2000;
      stop;
    (Z) :
      task s := 1000;
      stop;
enddecision;
When executing the previous code in the Telelogic Tau environment the second alternative is chosen. The reason is that the alternatives in the C code are not generated correctly which results in the execution of the first alternative instead of the second.

```c
switch (yVarP->RestartAddress) {
  case 0:
    (*(&yVarP->z011_s) = (SDL_Integer) 0);
    (yVarP->z012_t = 0);
    (yVarP->z012_t = 0);
    (yVarP->yDcn_z010_T = yVarP->z012_t);
    if (((yVarP->yDcn_z010_T) == (0))) {
      (yVarP->z011_s = 2000);
      xL_StopLabel: ;
      SDL_Stop(VarP);
      return;
    } else if (((yVarP->yDcn_z010_T) == (0))) {
      (yVarP->z011_s = 1000);
      goto xL_StopLabel;
    }
}
```

The previous example illustrates the difficulty of figuring out a similar test-case that would have revealed this compilation error and that SCEC is really effective in debugging such code generators.

**Conclusion** We have demonstrated in this chapter that our equivalence checking method does scale for industrial-sized software projects. In effect, the IEEE 802.11 MAC layer case study represents a typical application in that category. The second case study related to MIPv6 protocol is more academic but still gives an additional evidence of the applicability of our method.

The second major outcome is that the prototype tool that we have developed was able to discover a subtle bug in a commercial code generator. Such results are promising and clearly show that the development of equivalence checkers for software can be effective and rewarding as well.
Conclusion

This research shows that equivalence checking can successfully be used to verify the adherence of a software implementation against its specification. We addressed the case where the specification is described in terms of SDL whereas the implementation is coded in the C language. Both SDL and C have gained a wide acceptance in industry.

To demonstrate the effectiveness of our approach a tool called SCEC has been developed. As a result, developers can automatically verify the refinement of an abstract reference model into the final target code. By doing so, there is absolutely no need to re-validate again the target code obtained by using a compiler. The prototype SCEC was applied to practical cases from the telecommunication domain to prove the scalability of our method. In addition, SCEC allowed the discovery of a subtle bug in the commercial code generator. This result is very encouraging and shows that the effort spent in developing an equivalence checker does pay off.

One of the outcome of the thesis was to identify the hypotheses under which equivalence checking becomes algorithmically solvable and scalable as well. The main assumptions that underly this research are:

1. the restriction to a subset of SDL.
2. pre-defined knowledge of the compiler transformations.
3. transition-level scheduling enforcing the run-to-completion assumption.

To enable equivalence checking between SDL and C languages, a unifying representation was defined. In effect, SDL and C programs need first to be translated into an intermediate form. This intermediate representation that we called APN coeIces relevant aspects of both languages and is defined in terms of its syntax and operational semantics. An inductive argument based on bisimulation has been formalized for the sequential case. Two cases of the asynchronous parallel composition have been also treated, namely the server model and the activity thread model.
Scalability is achieved through cutpoints. Their usage is a key concept in our method since they allow to subdivide the verification problem into smaller sub-problems. We use cutpoints to break the loops as in Floyd’s method but also to relate the concrete program to its abstract version.

**Equivalence checking from the MDA perspective**  
UML is the most widely adopted modeling language by a large margin. UML allows developers to focus on levels of abstraction above implementation details. More generally, UML became a commonly accepted mean of expressing and communicating the basic concepts of a software design.

Taken to the highest level of abstraction, a domain model focuses not on software, but on the nature of the problem under consideration. Here, the model should use terms and symbols familiar to the people and systems of that particular application field. Currently, the industry is moving toward domain-specific languages dedicated to their respective area of use. More often, however, general-purpose modeling languages like UML are extended in standard ways to meet domain-specific modeling needs through profiles. Both approaches are consistent with the value of modeling in general: to provide abstractions for specifying problems and solutions in a more productive and effective manner. For instance, the International Telecommunication Union Recommendation Z.109 titled SDL Combined with UML incorporates the principal work on combining UML and SDL.

The investment in developing a model is compensated through the usage of model-to-code transformers. Ultimately, models should get executed and also translated automatically to implementations. Possibly to different implementation platforms. Correct model transformation poses therefore a real verification challenge. From that perspective, this research work promotes the use of fully automatic equivalence checkers to assert the correctness of the transformations. More research is required to examine further the kind of transformations necessary to derive efficient implementations from domain-specific languages and how to effectively build robust equivalence checkers for them. Guaranteeing the correctness of model transformations will remain of paramount importance in the model-centric approach.
Annex: Hypotheses

This annex presents the main assumptions underlying the implementation of the SCEC prototype.

9.1 SDL sub-set restriction

We depart from the SDL’96 Z.100 specification: The following constructs are not allowed.

1. Object oriented SDL
2. Procedures with states
3. Dynamic process creation
4. Remote procedure call
5. Macros
6. View/reveal
7. Service concept
8. Priority input
9. Unspecified number of process
10. External synonyms
11. Omission of parameters in signal input
12. Output via all
13. Timers with parameters
14. the upper limit of arrays must be given
15. Generator string is not supported
16. Generator PowerSet is not supported
17. the Any construct
9.2 SDL compiler

Concerning the compiler, we assume the following:

- the run-time environment definition is known. An example of such structure is given in section 3.5. This knowledge allows the re-construction of the extended finite state machine on the C side.

- Names of processes, states, connections, stimuli, procedures and operators are preserved by the compiler.

9.3 Scheduling

As far as the scheduling is concerned, the following must hold.

- the abstract and the concrete system use the same scheduler.

- processes are run to completion. This transition-level scheduling (i.e.; coarse-grained scheduling) guarantees, on the one hand, the absence of read/write conflicts between different processes, and on the other hand, that the correspondence links between the abstract and the concrete processes are maintained in a coherent way.
Bibliography


[29] R. Feldmann, B. Geppert, and F. Rossler. An integrating approach for developing distributed software systems-combining formal methods, software reuse, and the


Curriculum Vitae

Name : Malek Haroud
Profession : Applied Research Engineer STMicroelectronics
Diploma : MSc. Electrical Engineering EPFL
Ecole Polytechnique Fédérale de Lausanne
Date of birth : 20 July 1972
Nationality : Swiss
Languages : English, French, German, Arabic
Marital status : Married

Education
12/4/1995 MSc. Electrical Engineering -EPFL-
1989/1990 Cours de Mathématiques Spécial -EPFL-
1988/1989 Baccalauréat C, Lycée El Idrissi, Algiers, Algeria

Professional experience
09-1997/Present ST Microelectronics:
Cellular and wireless local area networks protocol design
08-1996/08-1997 ASCOM Business Systems Ltd:
Validation of GSM mobile phone protocols
05-1995/07-1996 BOSCH Telecoms:
MAC layer design for a TETRA Base-Transceiver station