Chapter 1

AN INTRODUCTION TO FORMAL METHODS

How do they apply in embedded system design?

Nikolaos S. Voros[1]
Wolfgang Mueller[2]
Colin Snook[3]

[1] INTRACOM S.A., Patra, Greece
[2] Paderborn University/C-LAB, Paderborn, Germany
[3]University of Southampton, Southampton, United Kingdom

Abstract: This chapter begins with an introduction to the main concepts of formal methods. Languages and tools for developing formal system models are also described, while the use of semi formal notations and their integration with formal methods is covered as well. At the end of the chapter, an overview of the current status of formal methods in embedded system design is presented.

Key words: Formal methods, formal languages, semi formal notations, tools for formal languages, embedded system design.

1. INTRODUCTION

Since we focus our investigations on formal specification and verification, we first give definitions of basic terms before introducing the theoretical background, formal languages and tools.

A *specification* can be regarded as a description that is intended to be as *precise*, *unambiguous*, *concise and complete* as possible in the context of its specific application [1]. A *formal specification* is a specification written in a formal language where a *formal language* is either based on a rigorous mathematical model or simply on a standardised programming or

specification language [2]. Due to its individual application, a formal specification can be (partly) *executable*. In most cases, formal specifications are for a mental execution by code review and for passing the specification around to members in a design team. In most cases only subsets of formal specification languages, e.g. of Z and VDM, are machine executable. A *formal method* implies the application of at least one formal specification language. Formal methods are often employed during system design when the degree of confidence in the prescribed system behaviour, extrapolated from a finite number of tests, is low. Formal methods are frequently applied in the design of ultra-reliable as well as complex concurrent or reactive systems. Formal specifications can be classified with respect to their specification style. Here, we can identify mainly two different classifications. One is mainly due to the field of programming languages; the other one comes from general systems specification.

1.1 Axiomatic vs. Denotational vs. Operational

In the context of the formal semantics of programming languages, we distinguish axiomatic, denotational and operational styles [3]. For *axiomatic semantics,* each language construct is specified in terms of axioms and deduction rules, where the Hoare style is a frequently applied notation [4]. *Denotational semantics* is a sort of functional semantics. Specification of denotational semantics consists of two main steps [5]. First, the syntactic domain, the syntactic constructs and the semantic domains are identified. Secondly, the semantic (valuation) functions mapping the syntax of a language to its semantics including its signatures are defined in detail. Valuation functions are usually given by a set of equations in form of typed λ-calculus notation [6]. *Operational semantics* finally defines a set of explicit commands changing the state of a more or less abstract machine. Operational semantics is usually close to the implementation of a language. Classical means for the definition of operational semantics are Finite State Machines, programming language-oriented notations, pseudo code and related means.

1.2 Model-based vs. Algebraic

Sommerville divides specification styles for non-trivial systems in *model-based* and *algebraic* approaches [7]. The model-based style gives the explicit specification of abstract machines. For a model-based specification, a system is defined in terms of mathematic entities (e.g. sets, relations, sequences). Operations are defined in terms of abstract states and how they affect those entities. The algebraic style specifies abstract data types in terms of axioms

defining relationships between its operations. This style is sometimes also denoted as *property-oriented* specification.

Due to recent trends in formal methods there is a huge number of formal methods, languages and available tools, each developed for a particular domain and purpose. It is impossible to give a complete enumeration of all known formal languages and tools for formal verification. We try to give a representative overview of widelly accepted and frequently applied approaches for formal specification and verification in the next sections. We first explore languages based on predicate logic, temporal logic, process algebras before we outline means, which do not fit in just one category like RSL, Action Semantics and Abstract State Machines. The final overview of formal verification tools mainly investigates model checkers and theorem provers.

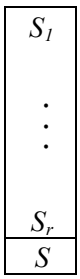
1.3 Theory

A huge array of formal specification means and verification tools is based on predicate logic so that we start with an overview of the theoretical background on predicate logic. *Predicate logic* is a branch of mathematical logic investigating the interpretation, construction and derivation of well-formed formulae (WFF). In first-order predicate logic, an atom has the form $P(t_1, ..., t_k)$, where P is a predicate symbol and t_1, ..., t_k are terms, which can be composed of function symbols, constant symbols and variables. *High-order predicate logic* is defined by additionally allowing the application of predicate symbols in terms. A well-formed formula is recursively composed of atoms, formulae, propositions or connectives (negation, disjunction, conjunction, implication, equivalence), conditions (equal, less, greater) and (universal and existential) quantifiers. Predicate logic is typed and multi-sorted. Predicate logic without predicates, functions and quantifiers is traditionally called *propositional logic*. In the context of predicate logic, we can distinguish model theory and proof theory [8].

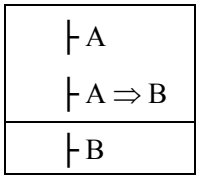
Model theory investigates the interpretation of syntactical sentences with respect to a model. The model is given by objects of a chosen domain. Syntactical sentences are composed of symbols, which are associated with objects by defining truth-valued functions that give the required interpretation. In predicate logic, well-formed formulae are composed of constant, predicate; function symbols are typically interpreted by Boolean-valued functions. A well-formed formula is *satisfiable* if there exists one interpretation that evaluates to true. An *axiom* is a well-formed formula that is true by definition. *Modal logic* considers the interpretation of formulae within different contexts (*worlds*) by so-called modal operators, such as *necessity* and *possibility*. *Temporal logic* is an instance of modal logic with

respect to time modalities [9]. Time modalities are defined by temporal future operators (e.g. Next, Henceforth, Eventually, Until, Unless) and past operators (e.g. Previous, Has-Always-Been, Once, Since, Back-To) [10]. Temporal operators typically refer to sequences of state transitions. Future-oriented temporal logic distinguishes *branching-time* and *linear-time* temporal logic. Branching-time temporal logic considers all possible paths within the tree defined by the set of state transitions, where the root of the tree is given by the current state. Linear-time temporal logic investigates just one path of that tree.

Proof theory investigates the relationship between sentences of a formal system using only rules for operating on the syntactical content of those sentences. A relation transforming a well-formed formula to another well-formed formula is denoted as an *inference rule*. An inference rule is composed of a set of sequents $S_1, ..., S_r$ and another sequent S, which follows from that set[1]:



A *sequent* is a pair whose first component is a set of formulae $A_1, ..., A_n$ (hypothesis) and its second component is a formula B (conclusion) that follows from the hypothesis: $A_1, ..., A_n \vdash B$
An example of an inference rule is the *modus ponens* which defines that if A holds and A implies B then B follows:



The application of inference rules is called a deduction. A deduction can be represented as a sequence $f_1, ..., f_n$ of formulae, where each f_i, $2 \leq i \leq n$, is deduced from f_{i-1} by the application of an inference rule. f_1 is an axiom and f_n is typically denoted as a theorem. A set of axioms and the set of all theorems derivable from the axioms by the application of all inference rules is called a *theory*. A theory is consistent if, and only if, it does not contain both *s*

[1] Sometimes also written as $\{S_1, ..., S_r\} \vdash S$

and $\neg s$ for all sentences s in the theory. Theorem provers are typically based on the sequent calculus and compute a deduction for a given set of axioms, theorems and inference rules.

A formal system of previous form is *decidable* if an algorithm can be constructed, which decides whether a given well-formed formula is a theorem. Propositional logic is decidable; first-order (and high-order) predicate logic is not decidable. When interpreting well-formed formulae, a formal system is *complete* if all true formulae can be derived. A formal system is *sound* if all derivable formulae are true. Propositional and first-order logic are sound and complete. High-order logic is sound and not complete.

2. LANGUAGES

To give a overview of formal specification languages, we have selected a representative set of languages from predicate logic, temporal logic, process algebras and a set of languages which do not fit into the previous categories.

2.1 Predicate Logic-Based Languages

In predicate-logic based specification languages, a system is usually defined by a set of functions, where each function is defined by its signature and a set of predicates that define pre- and post-conditions for the function. We introduce two classical predicate-logic based languages, Z and VDM, and the B language.

2.1.1 Z

Z was developed by the Programming Research Group at Oxford University and accepted as a BSI standard in 1989 [11]. Z is a specification language based on set theory with no official method. Object-Oriented and real-time extensions to Z are available as Object-Z and Timed Communicating Object-Z, respectively. There are conventions and practices to use Z as a model-based language. Though Z permits various specification styles, the state-based approach has been found the most convenient one in many applications. Z comes with a deductive system in order to reason about specifications. Z is based on typed set theory and first-order predicate logic. Invariants can be associated with a global state. Invariants relate pre- and post- conditions for operations. A Z specification is composed of modules (*schemas*). A Z schema is given by a schema name followed by its signature (set of domains) and a set of properties. Schemas support the separation of

the specification (and error conditions) from the system's behaviour. Schemas are combined by a schema calculus. For a model-based specification, the domains are typically defined by basic sets and abstract states are defined in terms of sets, relations, functions, sequences, etc. Properties are assigned to states by means of extra predicates, e.g. initial conditions for the initial state. Abstract operations define state transitions.

The wider acceptance of Z in recent years has advanced the available set of tools. Tools for assistance with proof, and tools for translating Z specifications to programming languages, are available. However, a first high-level Z specification is usually not machine-executable.

2.1.2 VDM

The Vienna Development Method (VDM) is a formal specification method with the model-based specification language VDM-SL (VDM Specification Language) [12]. VDM was initially developed for the formal description of PL/I at the IBM laboratory in Vienna. The VDM method considers the verification of step-wise refinement in the systems development process, i.e. data refinement and operation decomposition. VDM-SL is conceptually similar to Z. VDM specifications are based on logic assertions of abstract states (mathematic abstraction and interface specification). In contrast to Z, VDM uses keywords in order to distinguish the roles of different components while these structures are not explicit in Z. As with Z specifications, VDM specifications are usually not machine-executable. VDM supports the specification process by a mental execution with paper and pencil. However, proof assistance tools and tools for executing subsets are available.

2.1.3 B

B stands for a language, a method and tools [13]. The B language can considered to be a combination of Z and Pacal with some extensions for refinement. The B method is based on a hierarchical stepwise refinement and decomposition of a problem. After initial informal specification of requirements, an abstraction is made to capture, in a first formal specification, the most essential properties of a system. For example, these could be the main safety properties in a safety critical system. This top-level abstract specification is made more concrete and more detailed in steps, which may be one of two types. The specification can be refined either by changing the data structures used to represent the state information and/or by changing the bodies of the operations that act upon these data structures. Alternatively, the specification can be decomposed into subsections by

writing an implementation step that binds the previous refinement to one or more abstract machines representing the interfaces of the subsections. In a typical B project, many levels of refinement and decomposition are used to fully specify the requirements. Once a stage is reached when all the requirements have been expressed formally, further refinement and decomposition steps add implementation decisions until a level of detail is reached at level B_0, where code can be automatically generated for Ada and C++. B processing tools, like Atelier-B from Clearsy, are advanced theorem provers with code generation, which automatically provide theorems, i.e. proof obligations.

2.2 Temporal Logic-Based Languages

The next two paragraphs present temporal logic-based approaches: *Computational Tree Logic* and *Temporal Logic of Action*. The first one was developed in the context of classical model checking for hardware verification. The latter one was initially developed in the context of imperative programming languages.

2.2.1 CTL

CTL (Computational Tree Logic) was defined as a branching-time temporal logic for model checking. Several variations of CTLs are known for practical applications: *CTL*, *ACTL* and *CTL**. All CTLs are future-oriented. Only some approaches extend CTL with past modalities. CTL formulae express information about states or state transitions [14]. A *path* defines one possible future execution over state transitions beginning at the current state. All possible execution paths establish a tree with the current state at its root. The temporal operators are composed of two parts. The first part defines the quantification (path quantifier). It specifies either the truth of all paths starting from the current state (**A**) or specifies the existence of a path with certain properties (**E**). The second part specifies the ordering of events along one path or a set of paths by the operators: next-time (**X**), until (**U**) releases (**V**). *CTL** is a superset of *CTL* allowing the more general application of path quantifiers (**A** and **E**) in path formulae. *ACTL* is a subset of CTL eliminating the existence quantifier **E**. ACTL was defined to make the model checking more efficient. Linear-time temporal logic (LTL) investigates just one path in the tree rather than the complete tree.

2.2.2 TLA

TLA (Temporal Logic of Actions) is temporal logic-based theory providing a logic for specifying and reasoning about concurrent and reactive systems [15]. TLA+ is the language for writing TLA specifications. The corresponding tool for mechanically checking TLA proofs is TLP (TL Prover), which is based on the Larch theorem Prover (LP) and a BDD-based model checker. TLA supports the specification of refinements and checks properties like fairness. A TLA specification is a list of formulae. A TLA formula is constructed by negation, conjunction, disjunction, implication and equivalence including the existential quantifier and predicates.

2.3 Process Algebras

Another important class of formal specification languages is based on *Process Algebras*. A description based on process algebra generally defines a set of processes. A process p is sensitive on a set of events E_p and/or performs some actions A. In process algebras, a process p is identified by its set of partially order events, i.e. its *trace*. A *process algebra* is defined on processes and (mainly) compositional operators on processes, such as alternative, sequential and parallel compositioning. Some process algebras have labelling functions, others have events or actions.

Process algebras can also be considered as labelled transition systems, where a process p is transformed to a new process p' after accepting an event/action [16]. A transition system can be represented by a graph whose nodes are processes and edges are given by partially ordered state transitions. Edges are labelled by events/actions. Some approaches are based on the notion of *traces*. A *trace* is a sequence of actions/events reflecting the history of state transitions of a process. The behaviour of a process can be partially characterised by a set of traces. A large set of process algebras are derived from Milner's CCS (Calculus of Communicating Systems). A well-known ancestor is Hoare's CSP (Communicating Sequential Processes). Several parallel programming and specification languages, such as OCCAM and the ISO standard LOTOS are based on CSP, for instance. In the next paragraphs, we additionally investigate Circal, which extends CCS by some features for hardware verification.

2.3.1 CSS

CCS (Calculus of Communicating Systems) [17] specifies a system as a set of asynchronously running processes performing, possibly non-deterministic, actions. CCS allows processes to be guarded by actions

(*action-prefixing*). Processes are called agents in CCS and actions are referred to as the ports of an agent. Ports can be parameterised by free variables. Ports are divided in in- and out- port. Agents communicate by handshake via ports and are defined in terms of equations with a so-called agent expression on their right-hand side. Basic operators on agents are: prefixing, summation, composition, restriction, relabelling, simultaneous substitution and recursion. τ denotes the silent action and *1* the unit action. A set of equivalence relation supports the comparison of traces. A process logic PL (modal logic) covers the specification of properties over the set of traces in terms of formulae.

2.3.2 CSP

CSP (Communicating Sequential Processes) is conceptually similar to CCS [18]. CSP specifies a system as a set of asynchronously running processes acting on events. Processes communicate values (resp. events) via channels. To check CSP specification for properties, a satisfiability relation *sat* was introduced that allows CSP specifications to be checked with respect to a set of traces, i.e. specification vs. implementation. Hoare gives a listing of the differences between CCS and CSP in [18]. The main differences refer to the definition of non-determinism and the included correctness calculus. In CCS, non-determinism defined through τ is not associative in contrast to CSP. A major difference between CCS and CSP is in the correctness calculus. CCS defines a set of equivalences for observational behaviour, where CSP defines a satisfiability relation for specifying whether a process meets a given specification. CSP does not proof that an implementation does not satisfy a specification.

2.3.3 Circal

Circal (CIRcuit CALculus) is a process algebra for the formal verification of digital hardware including asynchronous hardware [19]. Circal defines a set of core operators and a set of derived laws. The laws are based on the semantics of the core operators using a labelled transition system and equivalence relations. Circal compares to CCS. In contrast to CCS (and CSP), Circal supports simultaneous actions and multiway composition. Circal allows processes to have guards with more than one action forming a *simultaneous guard,* which requires that all actions in the guard must occur simultaneously for the process to evolve. The Circal composition operator defines broadcast by allowing many processes to communicate via interconnected ports sharing a common label. This supports the modelling of bus structures. The combination of simultaneous

actions and multiway composition provides means for describing synchronous systems through the inclusion of a clock action in a simultaneous guard.

2.4 Other Approaches

We finally sketch three prominent representatives of approaches which do not exactly fit into the previous categories: RSL, Action Semantics and Abstract State Machines. They all integrate functional, imperative and concurrent aspects.

2.4.1 RSL

RSL Specification Language is a "wide-spectrum" specification and implementation language. It supports abstract, property-oriented specifications as well as low-level designs with concrete algorithms supporting specification refinement and reuse [20]. RSL is a combination of several specification languages, i.e. VDM, CSP, ACT-ONE and Standard ML. The basic concepts are inherited from VDM: type constructors for mapping, sets, cartesian products and function definitions (pre- and post-style). Concepts of CSP (resp. CCS) are incorporated to manage concurrency (channels, communication, concurrent composition). Class expressions, schemes and objects are the basic modules in RSL and similar to Standard ML functors. Basic class expressions are defined by theories (signatures and axioms). The algebraic specification, where types and values are constrained by the means of axioms, is based on the concepts of ACT-ONE.

2.4.2 Action Semantics

Action Semantics (AS) is a framework for the specification of the formal semantics of programming languages [21]. AS incorporates algebraic, functional (denotational) and imperative means. AS is a pragmatic approach based on the idea of functional semantics. An AS specification covers the specification of syntactic entities, semantic entities and semantic functions mapping syntactic entities to semantic entities. For semantic entities AS has *actions*, *data and yielders*. Data are static entities. Yielders represent dynamic data (unevaluated items of data) whose value depends on the current information. Actions are dynamic computational entities that define the behaviour. An action may diverge, complete, escape or fail. An action can be non-deterministic. Action notation supports the specification of control-flow, data-flow, process activation and communication. Non-commercial tools are available for interpreting and checking AS

specifications based on the principles of term rewriting (ASD Tools), as well as a AS to C compiler.

2.4.3 ASMs

Gurevich initially introduced Abstract State Machines (ASMs) as Evolving Algebras in 1991. A revised definition of with various extensions, commonly known as the Lipari Guide, was published in 1994 [22]. Whereas Gurevich has originally defined ASMs for considerations in complexity theory, multiple publications have demonstrated the applicability of ASMs for formal specification in various fields: hardware and software architecture, protocols, as well as programming and hardware description languages [23].

An ASM specification is a program executed on an abstract machine. The program comes in the form of rules. Rules are nested if–then–else clauses with a set of function updates in their body. Based on those rules, an abstract machine performs state transitions with algebras as states. Firing a set of rules in one step performs a state transition. ASMs are multi-sorted based on the notion of universes. Microsoft Research provides an executable version of ASMs, which execute ASMs under the .NET platform, i.e. AsmL [23].

3. TOOLS

Formal verification tools can roughly be classified in *equivalence checker*, *model checker* and *theorem prover* [14].

An *equivalence checker* compares two models for equivalence by applying different heuristics like reachability analysis, BBD equivalence or SAT-solving [14].

A *model checker* verifies a system against specified properties. The system is typically given as a collection of finite state machines. The property is usually defined in terms of temporal logic formulae [24].

For *theorem proving* the verification problem is given as a set of formulae and a set of inference rules. A theorem is then deduced from a set of axioms applying those rules. Theorem proving generally requires continuous user interactions where in most cases equivalence and model checking perform completely automatic verifications. The following gives a short overview of some model checkers and theorem provers.

3.1 Model Checkers

Model checking verifies a given set of state machines with respect to a set of temporal formulae, e.g. CTL or LTL. Many heuristics are applied to overcome the "state explosion" problem. We basically can distinguish explicit state checking and symbolic model checking. Explicit state checking is based on explicit state exploitation based on various breadth-first search and depth-first search heuristics. FDR, Spin and Murphi, are representatives of explicit state model checkers. Symbolic model checking is mostly based on the symbolic state representation by Binary Decision Diagrams (BDDs) [25] for efficient computation. BDD-based model checkers typically verify specifications with up to 200 state variables. Single systems with up to 10^{1300} reachable states (64 register ALU) have already been checked by the use of very dedicated abstraction techniques [24]. SMV is the classical BDD-based symbolic model checker, which many model checkers are based on, like the RuleBase from IBM and FormalCheck from Cadence. The following gives a short overview of a symbolic and a explicit state model checker, SMV and FDR.

SMV (Symbolic Model Verifier) from CMU [26] was the first model checker based on BDDs. The classical SMV checks finite state systems against CTL specifications using a OBDD-based (Ordered BDD) symbolic model checking algorithm with breath-first-search state traversal [25]. SMV checks for safety, liveness, fairness and deadlock freedom. A system is specified in the SMV input language. The input language allows the description of asynchronous as well as synchronous state systems, i.e. network of asynchronous, non-deterministic processes or Mealy machines. The language is restricted to finite data types. CV (CMU VHDL temporal logic model Checker) is a symbolic model checker for VHDL based on the concepts of SMV [27].

FDR (Failures-Divergence Refinement) is a Standard ML-based refinement checker and model checker for untimed CSP specifications and implementations [28]. FDR failures are sets of events that a process can refuse when being in a particular state. Divergences are states on which a process can perform an infinite sequence of actions. FDR checks for safety, liveness and refinements on the basis of traces. FDR provides 3 main functions: (1) transform specification into normal form, (2) divergence checking of the implementation and (3) model checking of specification and implementation. For checking states against traces, the transition system defined by a CSP specification is transformed into an equivalent normal form. This is mainly done to eliminate non-determinism and to make all actions visible. The model checker checks the validity of implementation states and failures with respect to the specification in normal form. The

check is based on a full and explicit expansion of the state space applying breadth-first search.

3.2 Theorem Prover

Theorem proving is the process of finding a proof for a given set of axioms and inference rules. There are two approaches to theorem proving which have reached wide acceptance and can be denoted as the classical approaches: *HOL* and the *Boyer-Moore Theorem Prover*. An alternative approach implements the State Delta Verification System (SDVS) of the Aerospace Corporation. SDVS is based on state delta logic, which is a variant of temporal logic. SDVS provides theorem proving for descriptions written in subsets of ISPS, Ada and VHDL [29].

In the following we first introduce the classical HOL and Boyer-Moore approach. Thereafter, we investigate the Prototype Verification System (PVS) and the Stanford Temporal Prover (STeP).

HOL is a model-based interactive proof assistant for high order logic [30]. The HOL approach is based on Milner's LCF (Logic for Computable Functions). In HOL, theorem proving is mainly applied for checking design refinements (implementation vs. specification). The specification is given as a set of axioms. The implementation represents the theorems that have to be interactively deduced from the axioms applying a set of specified inference rules. The logic of HOL is based on set theory and typed predicate logic. It extends first-order logic by equality, conditions and permits high-order functions (λ-expressions). Variables can range over functions, predicates and quantification over arbitrary types. All terms in high-order logic have a type. Basic HOL has the atomic types, boolean and natural numbers, as well as compound types, function types and a number of constants (e.g. implication and polymorphic equality). The constants come with a set of axioms and theorems. Temporal properties are defined as predicates on the execution. Predicates are given as typed sets. A small set of predefined inference rules support the deduction. Inference rules in HOL refer to functions, which return theorems if proper arguments are given. The system allows forward proofs as well as backward proofs based on the sequent calculus. In contrast to classical logics, theories are dynamic extendible objects in HOL.

BMTP (Boyer-Moore Theorem Prover) [30] is based on the principles of mathematic induction. Nqthm and ACL2 are both successors of BMTP [31,32]. In contrast to HOL, Nqthm provides a different style of theorem proving. Nqthm is a user-guided automatic deduction tool for checking properties of a specified system. The system has to be specified in terms of inductively defined total functions. Nqthm is based on propositional logic with equality. The basic theory includes axioms defining boolean

constants for true and false values, equality, if-then-else functions and Boolean arithmetic operations (conjunction, disjunction, negation, implication and equivalence). The theorem prover takes a term in propositional logic as input and reduces the term by the means of mathematic induction automatically generating induction schemes. Many heuristics and decision procedures are implemented as part of the transformation mechanism.

PVS (Prototype Verification System) from SRI (Stanford Research Institute) International Computer Science Laboratory is a theorem prover written in Standard ML [33]. The PVS specification language is based on high-order predicate logic. PVS specifications are divided into (parameterized) theories with assumptions, definitions, axioms and theorems. PVS expressions cover usual arithmetic and logical operators, function application, lambda abstraction and quantifiers. Constraints, such as the type of odd numbers, are introduced by the means of predicate subtypes and dependent types. The PVS theorem prover provides a collection of powerful primitive inference procedures that are interactively applied under user guidance based on a sequent calculus. The primitive inferences include propositional and quantifier rules, induction, rewriting and decision procedures. PVS integrates experimental integration of CTL model checking. Some inductive steps can be automatically discharged by the model checker.

STeP (Stanford Temporal Prover) integrates a model checker and an automatic deductive theorem prover [34]. The input is given as a set of temporal formulae and a transition system that is generated from a description in a reactive system specification language (SPL) or a description of a VHDL subset. The formulae and the transition system are the input for both, a model checker and an automatic theorem prover. The prover covers techniques based on first-order theorem proving, term rewriting, decision procedures and invariants. Interactive proofs are constructed with verification diagrams. STeP is for the verification of parameterized (N-component) circuit designs and parameterized (N-process) programs, including programs with infinite data domains.

4. SEMI-FORMAL NOTATIONS

In contrast to formal languages, semi-formal notations are notations that provide a set of symbols to represent specific roles in the description of a system, but have a loosely defined semantics. The use of a syntactically consistent notation generally brings a more formal feel to descriptions of systems than a natural language description would. This can be misleading

as the lack of a precise semantics leaves the description open to different interpretations.

4.1 The UML

The Unified Modelling Language [35] emerged as a standardisation of the leading object-oriented analysis and design methods that were competing for favour in the late 1980s and early 1990s. Responsibility for the standardisation has been taken over by an independent consortium, the Object Management Group (OMG). Several software tool manufacturers market tools to support the use of the UML.

The UML is a notation for use in modelling object-oriented designs and consists of 13 diagrams. The most important one for system specification and modelling are:

- *Use case diagrams* are a means of organizing requirements descriptions into event sequence scenarios. A scenario is triggered by an actor (an external object such as a person interacting with the system) and parts of the system's responsive actions are then packaged and represented by named symbols. The meaning of a particular symbol is defined textually, usually in natural language.
- *Class diagrams* are used to model the static structure of a problem or system. Entity types are represented by classes and the relationships between them are shown as associations and generalizations. Classes represent sets of like instances and are given attributes that represent state variables and values associated with each instance of the class. Classes also have operations that define how an instance's attributes and associations alter in response to events.
- *Collaboration diagrams* and *sequence diagrams* are similar to each other. They both show dynamic behaviour as objects (of the classes introduced in the class diagram) interacting, by passing messages or calling each other's operations, to perform a particular behaviour or task scenario. Sequence diagrams show the interaction as a time ordered sequence of messages passed between objects. Collaboration Diagrams show the same sequence of messages but overlaid on a network of connected objects rather than a time sequence. Note here, that UML 2.0 introduce several advanced features to sequence diagrams, like control structures by combined fragments, which are not available in collaboration diagrams.
- *State diagrams*[2] and *activity diagrams* – Statechart/Activity models, constructed and viewed via state diagrams and/or activity diagrams,

[2] In UML 2.0, state diagrams (and statecharts) are called state machines.

show behaviour in terms of a set of states and transitions between them. Each transition can be annotated with the event that causes it to occur, any guards, which must be true before it can occur and actions that are performed when it occurs. Activity diagrams are a development from state diagrams that also allow "forks" to activate more than one state simultaneously and synchronizations that require more than one state to be active before a transition can occur (when drawing activity diagrams, states are called activities). Statechart/Activity models can be used at several levels. For example, they can be attached to the logical model, to use cases or to classes. Note here, that UML 2.0 introduces a major revision to the unification of state, activity and sequence diagrams also covering Petri-Nets semantics.

4.2 Integrating Formal and Semi-Formal Notations

Semi-Formal Notations such as UML are gaining widespread popularity in industry but lack precision for describing detailed behaviour unambiguously. Conversely, formal notations have not gained widespread use in industry despite their recognized benefits. An integration of semi formal and formal notations may address the deficiencies of the semi formal notations while making the formal notation more approachable. Craigen, Gerhart and Ralston [36] found that better integration of formal methods with existing software assurance techniques and design processes was commonly seen as a major goal; they concluded that *successful integration is important to the long term success of formal methods*. Fraser, Kumar and Vaishnavi [37] discuss some of the reasons why this may be true and go on to describe a framework for classifying current formal specification processes according to the degree of transitional semiformal stages. The categories are direct (no transitional stages), sequential transitional (transitional stages developed prior to the formal specification) and parallel successive refinement (formal specification derived in parallel with semiformal specification through iterative process). Paige [38] analyses the composition of compatible notations and derives a meta-method for formal method (and semi-formal method) integration. Jackson [39] has developed a formal notation, Alloy and associated tool Alcoa. The Alloy notation has a partial graphical equivalent notation in which state can be expressed. This can then be converted into the textual version of the notation where operations can be added and analyses performed. Without tools to investigate the implications of different structures however, the graphical format is limited to illustration of structure. Several research groups have developed integration between graphical object-oriented notations, including the UML and formal notations such as B [13] and Z [40]. The precise UML

group[3] is a collaborative effort to precisely define UML semantics via formalization. The object constraint language, OCL [41] is a formal notation that is part of the UML. It can be used to attach formal constraint statements to elements of UML models to constrain their values. For example, the behaviour of an operation can be precisely defined by attaching OCL statements for the pre- and post-conditions of the operation.

5. APPLYING FORMAL METHODS IN SYSTEM DESIGN

The approaches reported in literature cover most of system design phases. Tools from industry and academia are also available for supporting the design of complex systems. They fall into the following categories:

- *Tools that support validation through simulation* Indicative examples are the SDL suite of Telelogic's Tau [42], which supports simulation of SDL system models and iLogix's Statemate MAGNUM [43] that allows simulation of Statechart models using specific test plans, while co-simulation among system models described in different specification languages is also available. A co-simulation approach using instruction set simulators, extended with the required dummy hardware models has been proposed for co-simulation of mixed hardware/software systems. CoWare [44] and Seamless [45] are typical co-design tools used in industry for hardware/software co-simulation and co-verification. The system description is given in VHDL or Verilog for hardware and C for software. Both allow co-simulation between hardware and software at the same abstraction level.
- *Tools supporting validation via test generation* AGATHA [46] belongs in this category and supports validation of system specifications. The system models can be described either in UML or in SDL or in Statecharts. Based on the system specifications, AGATHA automatically generates symbolic test cases, which are used as input to the initial system model in order to detect design weaknesses like deadlocks. Telelogic's TTCN suite [47] offers TTCN[4] testing of SDL system models and simulation of test suits (set of TTCN tests) referring to a specific test case. Esterel studio [48] allows specification and validation of complex systems using a validation

[3] Available at: *http://www.cs.york.ac.uk/puml/maindetails.html*.

[4] Tree and Tabular Combined Notation, or Testing and Test Control Notation in TTCN-3, is an ETSI standardized language for describing "black-box" tests for reactive systems such as communication protocols and services.

engine for checking the consistency of the implementation with the initial specifications, while it supports automatic test coverage generation for the system under design.

- *Tools that support formal verification* TNI-Valiosys consultant applies a formal verification technology, called Linear Programming Validation (LPV), at customers' models (mainly described in SDL) in order to validate and debug the behaviour of the SDL architectural model [49]. The purpose of this approach is to guarantee that the SDL model behaviour respects the specification requirements of the system. A similar approach has been adopted by imPROVE-HDL [49], a high performance model checker dedicated to hardware block validation, for white box and black box verification. The checker accepts as input a VHDL/Verilog description of the component and a description of the block's properties. The result is an indication of the block's compliance with the required properties, or a counter example to highlight problematic behaviours. For hardware implementations, SOLIDIFY [50] allows exhaustive functional verification of the system under design at the RTL level, aiming at reducing the test vectors needed for system level verification. The initial models of the system are described in a HDL like Verilog or VHDL.

The common denominator of most design practices is that despite the plethora of methods and tools supporting designers during various phases, most of them rely on simulation for system validation and verification. Simulation at early design stages and hardware/software co-simulation at late design stages are commonly used practices for validating and verifying the system under design.

In order to deal with the increasing verification complexity, there are attempts to apply formal methods for verifying system properties during every design phase. For example, B method/language for designing complex systems has been applied in [51,52], while B has also been applied for the design of hardware components. In [53], Event B is used to specify and refine a circuit, while the approaches in [54,55] model circuits close to an implementation level. The first one requires that the model uses basic logic gates which are modelled in terms of B machines; the second one allows higher data-types like integers in their models, while it introduces new structuring mechanisms into B which mirror those of VHDL closely.

The next book chapters introduce and describe thoroughly the *PUSSEE (Paradigm Unifying System Specification Environments for proven Electronic design) method*, which applies the main concepts for integrating formal and semi formal methods for embedded system design. In that context, the combined use of UML and B is introduced while a set of supporting tools is also described.

REFERENCES

1. U. Glässer, *Systems Level Specification and Modeling of Reactive Systems: Concepts, Methods and Tools*, Proceedings of EUROCAST 95, Springer Verlag, 1995.
2. IEEE, *Software Engineering Standards,* The Institute of Electrical and Electronics Engineers, 1987.
3. G. Winskel, *The Formal Semantics of Programming Languages*, The MIT Press, 1993.
4. C.A.R. Hoare, *An Axiomatic Basis for Computer Programming*, Communications of the ACM, 12(10), 1969.
5. J.E. Stoy, *Denotational Semantics: The Stoy-Strachey Approach to Programming Language Theory*, MIT Press, 1977.
6. H.P. Barendregt, *The Lambda Calculus, Its Syntax and Semantics*, North Holland, 1981.
7. I. Sommerville, Editor, *Software Engineering*, 4th Edition, Addison Wesley, 1992.
8. C.J. Hogger, *Essentials of Logic Programming*, Clarendon Press, 1990.
9. M. A. Orgun and W. Ma, *An Overview of Temporal and Modal Logic Programming*, Proceedings of First International Conference of Temporal Logic (ICTL 94), Springer Verlag, 1994.
10. Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*, Springer Verlag, 1992.
11. J. Bowen, *Specification and Documentation using Z: A Case Study Approach*, International Thomson Computer Press, 1996.
12. C.B. Jones, *Systematic Software Development using VDM*, Prentice Hall International, 1990.
13. J-R Abrial, *The B Book: Assigning Programs to Meanings*, Cambridge University Press, 1996.
14. Th. Kropf, *Introduction to Formal Hardware Verification*, Springer Verlag, 1998.
15. L. Lamport, *The Temporal Logic of Actions*, ACM Transactions on Programming Languages and Systems, 16(3), May 1994.
16. A. Arnold, *Finite Transition Systems*, Prentice Hall Int., 1994.
17. R. Milner, *Communication and Concurrency*, Prentice Hall, 1989.
18. C.A.R. Hoare, *Communicating Sequential Processes*, Prentice Hall, 1985.
19. G. J. Milne, *Circal and the Representation of Communication, Concurrency and Time*, ACM Transactions on Programming Languages and Systems, 7(2), 1985.
20. The RAISE Language Group, *The RAISE Specification Language*, Prentice Hall, 1992.
21. P.D. Mosses, *Action Semantics*, Number 26 in Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1992.
22. Y. Gurevich, *Evolving Algebras 1993: Lipari Guide*, In E. Börger, Editor, Specification and Validation Methods, Oxford University Press, 1994.
23. E. Börger and R. Stärk, *Abstract State Machines - A Method for High-Level System Design and Analysis*, Springer Verlag, 2003.
24. E. M. Clarke, O. Grumberg and D.A. Peled, *Model Checking*, MIT Press, 2000.
25. R. E Bryant, *Binary Decision Diagrams and Beyond Enabling Technologies for Formal Verification*, Proceedings of the 1995 IEEE/ACM Conference on Computer Aided Design, ACM Press, 1995.
26. K.L. McMillan, *The SMV System*, Available at: http://www-2.cs.cmu.edu /~modelcheck/smv.html, 2004.
27. CV, *A Model Checker for VHDL*, Available at: http://www-2.cs.cmu.edu /~cmuvhdl/index.html, 2004.
28. Formal Systems (Europe) Ltd, *FDR2 - User Manual and Tutorial*, Available at: http://www.fsel.com/documentation/fdr2/html/, 2003.

29. I. Filippenko, *Some Examples of Verifying Stage 3 VHDL Hardware Descriptions Using the State Delta Verification System (SDVS)*, Technical Report ATR-93(3738)-3, The Aerospace Corporation, 1993.
30. R. S. Boyer and J. S. Moore, *A Computational Logic*, Academic Press Inc, 1979.
31. R.S. Boyer and J. S. Moore, *A Computational Logic Handbook*, Academic Press Inc., 1988.
32. M. Kaufmann, P. Manolios, J.S. Moore, *How to Use ACL2*, Kluwer Academic Publishers, 2000.
33. J. Rushby, *The PVS Verification System*, Available at: http://www.csl.sri.com/sri-csl-pvs.html, 1995.
34. Z. Manna, A. Anuchitanukul, N. Borner, A. Browne, E. Chang, M. Colon, L. de Alfaro, H. Devarajan, H. Sipma and T. Uribe, *STeP the Stanford Temporal Prover*, Technical Report STAN-CS-TR-94-1518, Stanford University, 1994.
35. J. Rambaugh, I. Jacobson and G. Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1998.
36. D. Craigen, S. Gerhart and E. Ralston, *Formal Methods Reality Check: Industrial Usage*, IEEE Transactions on Software Engineering, vol. 21, No.2, 1995.
37. M. D. Fraser, K. Kumar and V. K. Vaishnavi, *Strategies for Incorporating Formal Specifications in Software Development*, Communications of the ACM, vol. 37, No. 10, 1994.
38. R. Paige, *Formal Method Integration via Heterogeneous Notations*, Ph.D Thesis, University of Toronto, 1997.
39. D. Jackson, *Alloy: A Lightweight Object Modelling Notation*, Technical Report 797, MIT Lab for Computer Science, 2000.
40. I. Houston and S. King, *CICS Project Report: Experiences and Results from the Use of Z in IBM*, Proceedings of The 4th International Symposium of VDM Europe. Vol. 1, Springer-Verlag, 1991.
41. J. Warmer and A. Kleppe, *The Object Constraint Language: Precise Modeling with UML*, Addison-Wesley, 1999.
42. SDL Suite, Available at: http://www.telelogic.com/products/tau/sdl/index.cfm, 2004.
43. Statemate, Available at: http://www.ilogix.com/products/magnum/index.cfm, 2003.
44. CoWare Inc, Available at: http://CoWare N2C Method Manual. Version 3.1, 2001.
45. Seamless, Available at: http://www.mentor.com/seamless/, 2003.
46. D. Lugato et al, *Validation and Automatic Test Generation on UML Models: The AGATHA Approach*, Electronics Notes in Theoretical Computer Science 66 No.2, 2002.
47. TTCN Suite, Available at: http://www.telelogic.com/products/tau/ttcn/index.cfm, 2003.
48. Esterel Studio. Available at: http://www.esterel-technologies.com/v3/, 2003.
49. Valiosys, Available at: http://www.tni-valiosys.com/, 2003.
50. Solidify, Available at: http://www.saros.co.uk/, 2003.
51. J. Draper et al, *Evaluating the B method on an avionics example*, Proceedings of Data Systems in Aerospace (DASIA) Conference, 1996.
52. C. Snook, L. Tsiopoulos and M. Walden, *A Case Study in Requirement Analysis of Control Systems using UML and B*, Proceedings of International Workshop on Refinement of Critical Systems, Methods, Tools and Developments, 2003.
53. J.-R. Abrial, *Event Driven Electronic Circuit Construction*, Available at: http://www.atelierb.societe.com/ressources/articles/cir.pdf
54. S. Hallestrede, *Parallel Hardware Design in B*, Proceedings of 3rd International Conference of B and Z Users, Lecture Notes in Computer Science, vol. 2651, Springer-Verlag, 2003.
55. W. Ifill et al, *The Use of B to Specify, Design and Verify Hardware*, In High Integrity Software, Kluwer Academic Publishers, 2001.