6

System Validation

J.V. Kapitonova,
A.A. Letichevsky, and
V.A. Volkov
Glushkov Institute of Cybernetics, National Academy of Science of Ukraine

T. Weigert
Global Software Group, Motorola

6.1 Introduction

Toward the end of the 1960s system designers and software engineers faced what was then termed as "software crisis." This crisis was the direct outcome of the introduction of a new generation of computer hardware. The new machines were substantially more powerful than the hardware available until then, making large applications and software systems feasible. The strategies and skills employed in building software for the new systems did not match the new capabilities provided by the enhanced hardware. The results were delayed projects, sometimes for years, considerable cost overruns, and unreliable applications with poor performance. The need arose for new techniques and methodologies to implement large software systems. The now classic "waterfall" software life-cycle model was then introduced to meet these needs.

The 1960s have long gone by, but the software crisis still remains. In fact the situation has worsened — the implementation disasters of the 1960s are being succeeded by design disasters. Software systems have reached levels of complexity at which it is extremely difficult to arrive at complete, or even consistent, specifications and it is nearly impossible to know all the implications of one's requirement decisions. Further, the availability of hardware and software components may change during the course of the development of a system, forcing a change in the requirements. The customer may be unsure of the requirements altogether. The situation is even worse for embedded systems: the real time and distributed aspects of such systems impose additional design difficulties and introduces the possibility of concurrency

pathologies such as deadlock or lifelock, resulting from unforeseen interactions of independently executing system components. A number of new methodologies, such as rapid prototyping, executable specifications, and transformational implementation have been introduced to address these problems in order to arrive at shorter cycle time and increased quality of the developed systems. Although each of these methodologies addresses different concerns they share the underlying assumption that verification and validation be performed as close to the customer requirements as possible.

While verification tries to ensure that the system is built "right," that is, without defects, validation attempts to ensure that the "right" system is developed, that is, a system that matches what the customer actually wants. The customer needs is captured in the system requirements. Many studies have demonstrated that errors in system requirements are the most expensive as they are typically discovered late, when one first interacts with the system; in the worst case such errors can force complete redevelopment of the system.

In this chapter, we examine techniques aimed at discovering the unforeseen consequences of requirements as well as omissions in requirements. Requirements should be consistent and complete. Roughly speaking, consistency means the existence of an implementation that meets the requirements; completeness means that the implementation (its function or behavior) is defined uniquely by the requirements. Validation of a system is to establish that the system requirements are consistent and complete.

Embedded systems [1–3] consist of several components that are designed to interact with one another and with their environment. In contrast to functional systems, which are specified as functions from input to output values, an embedded system is defined by its properties. A property is a set of desired behaviors that the system should possess.

In Section 6.2, we present a mathematical model of embedded systems. Labeled transition systems, representing the environment and agents inserted into this environment by means of a continuous insertion function are used for representing system requirements at all levels of details. Section 6.3 presents a survey of freely available systems that could be used to validate embedded systems as well as references to commercially available systems. In Section 6.4, we present a notation to describe the requirements of embedded systems. Two kinds of requirements are distinguished: *static requirements* define the properties of system and environment states and the insertion function of the environment; *dynamic requirements* define the properties of histories and behavior of system and environment. Hoare-style triples are used to formulate static requirements; logical formulae with temporal constraints are used to formulate dynamic requirements. To define transition systems with a complex structure of states we rely on *attributed transition systems* which allow to split the definition of a transition relation into a definition of transitions on a set of attributes, and formulate general transition rules for the entire environment states. We also present a tool for reasoning about the embedded systems and discuss more formally the consistency and completeness condition for a set of requirements. Our approach does not require the developers to build software prototypes, which are traditionally used for checking consistency of a system under development. Instead, one develops formal specifications and uses proofs to determine consistency of the specification. Finally, Section 6.5 presents the specification of a simple scheduler as a case study and reports the results on applying these techniques to the systems in various application domains.

We have observed the following time distribution in the software development cycle: 40% of the cycle time is spent on requirements capture, 20% on coding, and 40% on testing. Requirements capture, includes not only the development of requirements but also their corrections and refinement during the entire development cycle. According to Brooks [4], 15% of the development efforts are spent on validation, that is, ensuring that the system requirements are correct. Therefore, improving validation has a significant impact on development time, even for successful requirement specifications. For failed requirements that forced major system redevelopment, the impact is obviously much higher.

6.2 Mathematical Models of Embedded Systems

In the embedded domain, the main properties of the systems concern the interaction of components with each other and with the environment. The primary mathematical notion to formally represent interacting

and communicating systems is that of a *labeled transition system*. When formulating requirements or developing high-level specifications we are not interested in the internal structure of the states of a system and consider these states, and therefore also the systems, as identical up to some equivalence. The abstraction afforded by this equivalence leads to the general notion of an *agent* and its *behavior*. Agents exist in some environment and an explicit definition of the interaction of agents and environments in terms of a function that embeds the agent in this environment helps to specialize the mathematical models to particular characteristics of the subject domain.

6.2.1 Transition Systems

The most general abstract model of software and hardware systems, which evolve in time and change states in a discrete way, is that of a discrete dynamic system. It is defined as a set of states and a set of histories, describing the evolution of a system in time (either discrete or continuous). As a special case, a *labeled transition system* over the set of actions A is a set S of states together with the transition relation $T \subseteq S \times A \times S$. If $(s, a, s') \in T$, we usually write this as $s \xrightarrow{a} s'$ and say that a system S moves from the state s to state s' while performing the action a. (Sometimes the term "event" is used instead of "action.") An automaton is a more special case, where the set of actions is the set of input/output values. Continuity of time, if necessary, can be introduced by supplying actions with a duration, that is, by considering complex actions (a, t), where a is a discrete component of an action (its content) and t is a real number representing the duration of a. In timed automata, duration is defined nondeterministically and intervals for possible durations are used instead of specific moments in time.

Transition systems separate the observable part of a system, which is represented by actions, from the hidden part, which is represented by states. Actions performed by a system are observable by an external observer and other systems, which can communicate with the given system, synchronizing their actions, and combining their behaviors. The internal states of a system are not observable; they are hidden. Therefore, the representation of states can be ignored when considering the external behavior of a system.

The activity of a system can be described by its history which is a sequence of transitions, beginning from an initial state:

$$s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \cdots s_n \xrightarrow{a_{n+1}} s_{n+1} \cdots$$

A history can be finite or infinite. Each history has an observable part (a sequence of actions $a_1, a_2, \ldots, a_n, \ldots$) and a hidden part (a sequence of states). The former is called a *trace* generated by the initial state s_0 (in Reference 5, the term *behavior* is used instead of trace). Two states are called to be *trace-equivalent* if the set of all traces generated by these states coincide.

A *final* history cannot be continued: it is infinite or for the last state s_n in the sequence, there are no transitions $s_n \xrightarrow{a_n} s_{n+1}$ from this state; such a state is called a *final* state. We distinguish a final state representing *successful termination* from *deadlock* states (states where one part of a system is waiting for an event caused by another part and the latter is waiting for an event caused by the former) and *divergent* or *undefined* states. Such states can be defined later or constitute *livelocks* (states that contain hidden infinite loops or infinite recursive unfolding without observable actions).

Transition systems can be nondeterministic in which a system can move from a given state s into different states performing the same action a. A labeled transition system (without hidden transitions) is *deterministic* if for arbitrary transitions $s \xrightarrow{a} s'$ and $s \xrightarrow{a} s''$, it follows that $s' = s''$, and that there are no states representing both successful termination and divergence.

To define transition systems with a complex structure of states we rely on *attributed transition systems*. If e is a state of an environment and f is an attribute of this environment, then the value of this attribute will be denoted as $e \cdot f$. We will represent a state of an environment with attributes $f_1, \ldots, f_n$ as an object with public (observable) attributes $f_1 : t_1, \ldots, f_n : t_n$, where $t_1, \ldots, t_n$ are types, and some hidden private part.

6.2.2 Agents

Agents are objects that can be recognized as separate from the "rest of the world," that is, other agents or the environment. They change their internal state and can interact with other agents and the environment, performing observable actions. The notion of an *agent* formalizes such diverse objects as software components, programs, users, clients, servers, active components of distributed systems, and so on.

In mathematical terms, agents are labeled transition systems with states considered up to bisimilarity. We are not interested in the structure of the internal states of an agent but only in its observable behavior. The notion of an agent as a transition system considered up to some equivalence has been studied extensively in concurrency theory; van Glabbeek [6] presents a survey of the different equivalence relations that have been proposed to describe concurrent systems. These theories use an algebraic representation of agent states and develop a corresponding algebra so that equivalent expressions define equivalent states. The transition relation is defined on the set of algebraic expressions by means of rewriting rules and recursive definitions.

Some representations avoid the notion of a state, and instead, if for some agent E a transition for action a is defined, it is said that the agent performs the action a and thus becomes another agent E'.

6.2.2.1 Behaviors

Agents with the same behavior (i.e., agents which cannot be distinguished by observing their interaction with other agents and environments) are considered equivalent. We characterize the equivalence of agents in terms of the complete continuous algebra of behaviors $F(A)$. This algebra has two sorts of elements — behaviors $u \in F(A)$, represented as finite or infinite trees, and actions $a \in A$, and two operations — *prefixing* and *nondeterministic choice*. If a is an action and u is a behavior, prefixing results in a new behavior denoted as $a \cdot u$. Nondeterministic choice is an associative, commutative, and idempotent binary operation over behaviors denoted as $u + v$, where $u, v \in F(A)$. The neutral element of nondeterministic choice is the deadlock element (impossible behavior) 0. The empty behavior Δ performs no actions and denotes the successful termination of an agent. The generating relations for the algebra of behaviors are as follows:

$$u + v = v + u$$

$$(u + v) + w = u + (v + w)$$

$$u + u = u$$

$$u + 0 = u$$

$$\emptyset \cdot u = 0$$

where $\emptyset$ is the impossible action.

Both operations are continuous functions on the set of all behaviors over A. The approximation relation $\subseteq$ is a partial order with minimal element $\perp$. Both prefixing and nondeterministic choice are monotonic with respect to this approximation:

$$\perp \subseteq u$$

$$u \subseteq v \Rightarrow u + w \subseteq v + w$$

$$u \subseteq v \Rightarrow a \cdot u \subseteq a \cdot v$$

The algebra $F(A)$ is constructed so that prefixing and nondeterministic choice are also continuous with respect to the approximation and it is closed relative to the limits (least upper bounds) of the directed sets of finite behaviors. Thus, we can use the fixed point theorem to give a recursive definition of behaviors starting

from the given behaviors. Finite elements are generated by three termination constants: Δ (successful termination), $\perp$ (the minimal element of the approximation relation), and 0 (deadlock).

$F(A)$ can be considered as a transition system with the transition relation defined by $u \xrightarrow{a} v$ if u can be represented in the form $u = a \cdot v + u'$. The terminal states are those that can be represented in the form $u + \Delta$, divergent states are that which can be represented in the form $u + \perp$. In algebraic terms we can say that u is terminal (divergent) iff $u = u + \Delta$ ($u = u + \perp$), which follows from the idempotence of nondeterministic choice. Thus, behaviors can be considered as states of a transition system. Let beh(s) denote the behavior of an agent in a state s, then the behavior of an agent in state s can be represented as the solution $u_s \in F(A)$ of the system

$$u_s = \sum_{s \xrightarrow{a} t} a \cdot u_t + \varepsilon_s \tag{6.1}$$

where $\varepsilon_s = 0$ if s is neither terminal nor divergent, $\varepsilon_s = \Delta$ if s is terminal but not divergent, $\varepsilon_s = \perp$ for divergent but not terminal states, and $\varepsilon_s = \Delta + \perp$ for states which are both terminal and divergent. If all summands in the representation (6.1) are different, then this representation is unique up to associativity and commutativity of nondeterministic choice.

As an example, consider the behavior u defined as $u = \mathbf{tick}.u$. This behavior models a clock that never terminates. It can be represented by a transition system with only one state u which generates the infinite history

$$u \xrightarrow{\texttt{tick}} u \xrightarrow{\texttt{tick}} \cdots$$

The infinite tree with only one path representing this behavior can be obtained as the limit of the sequence of finite approximations $u^{(0)} = \perp$, $u^{(1)} = \mathbf{tick}.\perp$, $u^{(2)} = \mathbf{tick.tick}.\perp, \ldots$. Now consider,

$$u = \mathbf{tick}.u + \mathbf{stop}.\Delta$$

This is a model of a clock which can terminate by performing the action **stop**, but the number of steps to be done before terminating are not known in advance. The transition system representing this clock has two states, one of which is a terminal state. The first two approximations of this behavior are

$$u^{(1)} = \mathbf{tick}.\perp + \mathbf{stop}.\Delta$$
$$u^{(2)} = \mathbf{tick}.(\mathbf{tick}.\perp + \mathbf{stop}.\Delta) + \mathbf{stop}.\Delta$$

Note that, the second approximation cannot be written in the form $\mathbf{tick.tick}.\perp + \mathbf{tick}.\perp + \mathbf{stop}.\Delta$ because distributivity of choice does not hold in behavior algebra.

$$u = \mathbf{tick}.u + \mathbf{tick}.0$$

describes a similar behavior but is terminated by deadlock rather than successfully.

6.2.2.2 Bisimilarity

Trace equivalence is too weak to capture the notion of the behavior of a transition system. Consider the systems shown in Figure 6.1.

Both systems in Figure 6.1 start by performing the action a. But the system at the left-hand side has a choice at the second step to perform either action b or c. The system on the right can only perform an action b and can never perform c or it can only perform c and never perform b, depending on what decision was made at the first step. The notion of *bisimilarity* [7] captures the difference between these two systems.

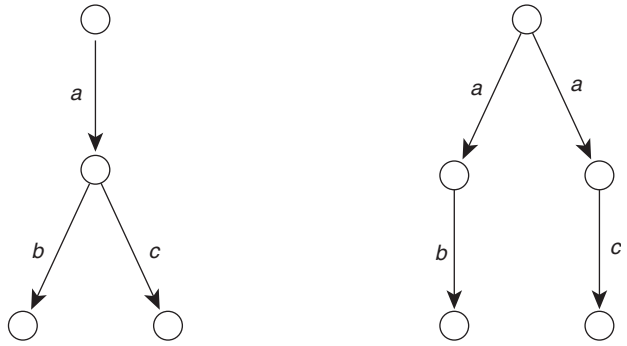

FIGURE 6.1 Two systems which are trace equivalent but have different behaviors.

A binary relation $R \subseteq S \times S$ on the set of states S of a transition system without terminal and divergent states is called a bisimulation if for each s and t such that $(s, t) \in R$ and for each $a \in A$:

1. If $s \xrightarrow{a} s'$ then there exists $t' \in S$ such that $t \xrightarrow{a} t'$ and $(s', t') \in R$.
2. If $t \xrightarrow{a} t'$ then there exists $s' \in S$ such that $s \xrightarrow{a} s'$ and $(s', t') \in R$.

Two states s and t are called *bisimilar* if there exists a bisimulation relation R such that $(s, t) \in R$. Bisimilarity is an equivalence relation whose definition is easily extended to the case when R is defined as a relation between the states of two different systems, considering the disjoint union of their sets of states. Two transition systems are bisimilar if each state of one of them is bisimilar to some state of the other.

For systems with nontrivial sets of terminal states S_Δ and divergent states $S_\perp$, *partial bisimulation* is considered instead of bisimulation. A binary relation $R \subseteq S \times S$ is a partial bisimulation if for all s and t such that $(s, t) \in R$ and for all $a \in A$,

1. If $s \in S_\Delta$ then $t \in S_\Delta$ and if $s \notin S_\perp$ then $t \notin S_\perp$.
2. If $s \xrightarrow{a} s'$ then there exists t' such that $t \xrightarrow{a} t'$ and $(s', t') \in R$.
3. If $t \xrightarrow{a} t'$ then there exists s' such that $s \xrightarrow{a} s'$ and $(s', t') \in R$.

A state s of a transition system S is called a *bisimilar approximation* of t, denoted by $s \subseteq_B t$, if there exists a partial bisimulation R such that $(s, t) \in R$. *Bisimilarity* $s \sim_B t$ can then be introduced as the relation $s \subseteq_B t \wedge t \subseteq_B s$. For attributed transition systems, the additional requirement is that if $(s, t) \in R$, then s and t have the same attributes.

A divergent state without transition approximates arbitrary other states that are not terminal. If s approximates t and s is convergent (not divergent) then t is also convergent, s and t have transitions for the same sets of actions, and satisfy the same conditions as for bisimulation without divergence. Otherwise if s is divergent, the set of actions, for which s has transitions, is only included in the set of actions for which t has transitions, that is, s is less defined than t. For the states of a transition system it can be proved that

$$s \subseteq_B t \;\Leftrightarrow\; \mathrm{beh}(s) \subseteq \mathrm{beh}(t)$$

$$s \sim_B t \;\Leftrightarrow\; \mathrm{beh}(s) = \mathrm{beh}(t)$$

and, therefore, the states of an agent considered up to bisimilarity can be identified with corresponding behaviors. If S is a set of states of an agent then a set $U = \{\mathrm{beh}(s) | s \in S\}$ is a set of all its behaviors. This set is *transition closed* which means that $u \in U$ and $u \xrightarrow{a} v$ implies $v \in U$. Therefore, U is also a transition system equivalent to S and can be used as a standard behavior representation of an agent.

For many applications, a weaker equivalence such as weak bisimilarity introduced by Milner [8], or insertion equivalence as discussed in Section 6.2.3, have been considered. Note that, for deterministic systems, if two systems are trace-equivalent, they are also bisimilar.

6.2.2.3 Composition of Behaviors

Composition of behaviors is defined as an operation over agents and is expected to preserve equivalence; it can, therefore, also be defined as an operation on behaviors.

The *sequential composition of behaviors* u and v is a new behavior denoted as $(u; v)$ and defined by means of the following inference rules and equations:

$$\frac{u \xrightarrow{a} u'}{(u; v) \xrightarrow{a} (u'; v')} \tag{6.2}$$

$$((u + \Delta); v) = (u; v) + v \tag{6.3}$$

$$((u + \bot); v) = (u; v) + \bot \tag{6.4}$$

$$(0; u) = 0 \tag{6.5}$$

We consider a transition system with states built from arbitrary behaviors over the set of states A by means of operations of the behavior algebra $F(A)$ and a new operation denoted as $(u; v)$. Expressions are considered up to the equivalence defined by the above equations (thus, the extension of a behavior algebra by this operation is conservative). The inference rule (6.2) defines a transition relation on a set of equivalence classes.

From rule (6.2) and equation (6.4) it follows that $(\Delta; v) = v$ and $(\bot; v) = \bot$. One can prove that $(u; \Delta) = u$ and that sequential composition is associative and distributives to the left

$$((u + v); w) = (u; w) + (v; w)$$

Sequential composition can also be defined explicitly by the following recursive definition:

$$(u; v) = \sum_{u \xrightarrow{a} u'} a \cdot (u'; v) + \sum_{u = u + \varepsilon} (\varepsilon; v)$$

6.2.2.3.1 Parallel Composition of Behaviors

We define an algebraic structure on the set of actions A by introducing the combinator $a \times b$ of actions a and b. This operation is commutative and associative with the impossible action $\emptyset$ as the zero element ($a \times \emptyset = \emptyset$). As $\emptyset \cdot u = 0$, there are no transitions labeled $\emptyset$. The inference rules and equations defining the parallel composition $u \parallel v$ of behaviors u and v are

$$\frac{u \xrightarrow{a} u', v \xrightarrow{b} v', a \times b \neq \emptyset}{u \parallel v \xrightarrow{a \times b} u' \parallel v'}$$

$$\frac{u \xrightarrow{a} u'}{u \parallel v \xrightarrow{a} u' \parallel v}$$

$$\frac{u \xrightarrow{a} u'}{u \parallel (v + \Delta) \xrightarrow{a} u'}$$

$$\frac{v \xrightarrow{a} v'}{u \parallel v \xrightarrow{a} u \parallel v'}$$

$$\frac{v \xrightarrow{a} v'}{(u + \Delta) \parallel v \xrightarrow{a} v'}$$

$$(u + \Delta) \parallel (v + \Delta) = (u + \Delta) \parallel (v + \Delta) + \Delta$$

$$(u + \bot) \parallel v = (u + \bot) \parallel v + \bot$$

$$u \parallel (v + \bot) = u \parallel (v + \bot) + \bot$$

The following equations for termination constants are direct consequences of these definitions:

$$\Delta \parallel \varepsilon = \varepsilon \parallel \Delta = \varepsilon \quad \bot \parallel \varepsilon = \varepsilon \parallel \bot = \bot$$

$$0 \parallel \varepsilon = \varepsilon \parallel 0 = 0 \quad \text{if } \varepsilon \neq \varepsilon + \bot$$

$$0 \parallel \varepsilon = \varepsilon \parallel 0 = \bot \quad \text{if } \varepsilon = \varepsilon + \bot$$

Parallel composition is commutative and associative.

Parallel composition is the primary means for describing the interaction of agents. The simplest interaction is interleaving, which trivially defines composition as $a \times b = \emptyset$ for arbitrary actions. Agents in a parallel composition interact with each other and can synchronize via combined actions. Parallel composition can also be defined explicitly by the following recursive definition:

$$(u \parallel v) = \sum_{\substack{u \xrightarrow{a} u' \\ v \xrightarrow{b} v'}} (a \times b) \cdot (u' \parallel v') + \sum_{u \xrightarrow{a} u'} a \cdot (u' \parallel v) + \sum_{v \xrightarrow{b} v'} b \cdot (u \parallel v') + \varepsilon_u \parallel \varepsilon_v$$

where ε_u is a termination constant in the equational representation of behavior u.

6.2.3 Environments

An *environment* E is an agent over an action algebra C with an *insertion function*. All states of the environment are initial states. The insertion function, denoted by $e[u]$ takes an argument e (the behavior of an environment) and the behavior of an agent over an action algebra A in a given state u (the action algebra of agents may be a parameter of the environment) and yields a new behavior of the same environment. The insertion function is continuous in both its arguments.

We consider agents up to a weaker equivalence than bisimilarity. Consider the example in Figure 6.2. Clearly, these systems are not bisimilar. However, if a represents the transmition of a message, and b represents the reception of that message, the second trace on the left-hand side figure would not be possible within an environment that supports asynchronous message passing. Consequentially, both systems would always behave the same. Insertion equivalence captures this difference: the environment can impose constraints on the inserted agent, such as disallowing the behavior $b \cdot a$, in this example. In such environment, both behaviors shown in Figure 6.2 are considered equivalent.

Insertion equivalence depends on the environment and its insertion function. Two agents u and v are insertion equivalent with respect to an environment E, written as $u \sim_E v$, if for all $e \in E$, $e[u] = e[v]$. Each agent u defines a transformation on the set of environment states; two agents are equivalent with respect to a given environment if they define the same transformation of the environment.

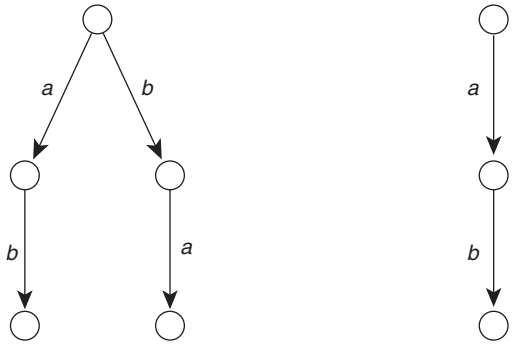

FIGURE 6.2 Two systems which are not bisimilar, but may be insertion equivalent.

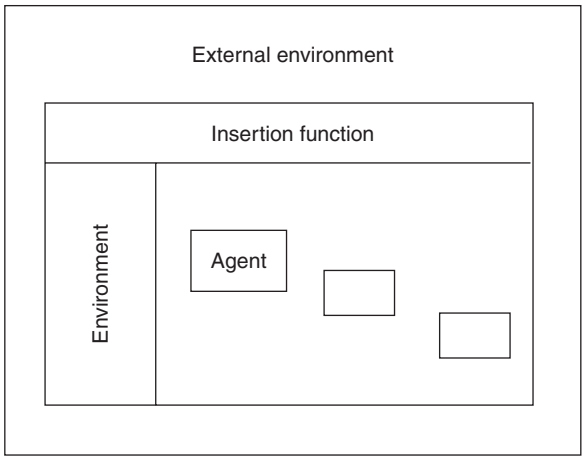

FIGURE 6.3 Agents in environment.

After insertion of an agent into an environment, the new environment is ready to accept new agents to be inserted. Since insertion of several agents is a common operation, we shall use the notation

$$e[u_1, \ldots, u_n] = e[u_1] \cdots [u_n]$$

as a convenient shortcut for insertion of several agents.

In this expression, $u_1, \ldots, u_n$ are agents inserted into the environment simultaneously, but the order of insertion may be essential for some environments. If we wanted an agent u to be inserted after an agent v, we must find some transition $e[u] \xrightarrow{a} s$ and consider the expression $s[v]$. Some environments can move independently, suspending the actions of an agent inserted into it. In this case, if $e[u] \xrightarrow{a} e'[u]$, then $e'[u, v]$ describes the simultaneous insertion of u and v into the environment in state e' as well as the insertion of u when the environment is in a state e and is followed by the insertion of v.

An agent can be inserted into the environment $e[u_1, u_2, \ldots, u_n]$, or that environment can itself be considered as an agent which can be inserted into a new external environment e' with a different insertion function. An environment with inserted agents as a transition system is considered up to bisimilarity, but after insertion into a higher level environment it is considered up to insertion equivalence (Figure 6.3).

Some example environments arising in real-life situations are:

- A vehicle with sensors is an environment for a computer system.
- A computer system is an environment for programs.

- The operating system is an environment for application programs.
- A program is an environment for data, especially when considering interpreters or higher-order functional programs.
- The web is an environment for applets.

6.2.3.1 Insertion Functions

Each environment is defined by its insertion function. The restriction on the insertion function to be continuous is too weak and in practice more restricted types of insertion functions are considered. The states of environments and agents can be represented in algebraic form as expressions of a behavior algebra. To define an insertion function it is sufficient to define transitions on the set of expressions of the type $e[u]$. We use rules in the form of rewriting logic to define these transitions. The typical forms of such rules are:

$$F(x)[G(y)] \rightarrow d \cdot F'(z)[G'(z)]$$

$$F(x)[G(y)] \rightarrow F'(z)[G'(z)]$$

where $x = (x_1, \ldots, x_n)$, $y = (y_1, \ldots, y_n)$, $z = (x_1, x_2, \ldots, y_1, y_2, \ldots)$, $x_1, x_2, \ldots, y_1, y_2$ are action or behavior variables, F, G, F', G' are expressions in the behavior algebra, that is, expressions built by nondeterministic choice and prefixing. More complex rules allow arbitrary expressions on the right-hand side in the behavior algebra extended by insertion as two sorted operation. The first type of rule defines observable transitions

$$F(x)[G(y)] \xrightarrow{d} F'(z)[G'(z)]$$

The second type of rule defines unlabeled transitions which can be used as auxiliary rules. They are not observable outside the environment and can be reduced by the rule

$$\frac{e[u] \xrightarrow{*} e'[u'], e'[u'] \xrightarrow{d} e''[u'']}{e[u] \xrightarrow{d} e''[u'']}$$

where $\xrightarrow{*}$ means the transitive closure of unlabeled transitions. Special rules or equations must be added for termination constants. Rewriting rules must be left linear with respect to the behavior variables, that is, none of the behavior variables can occur more than once in the left-hand side. Additional completeness conditions must be present to ensure all possible states of the environment are covered by the left-hand side of the rules. Under these conditions, the insertion function will be continuous even if there are infinitely many rules. This is because, to compute the function $e[u]$ one needs to know only some finite approximations of e and u. If e and u are defined by means of a system of fixed point equations, these approximations can be easily constructed by unfolding these equations sufficiently many times.

Insertion functions that are defined by means of rewriting rules can be classified on the basis of the height of terms $F(x)$ and $G(y)$ in the left-hand side of the rules. The simplest case is when this height is no more than 1, that is, terms are the sum of variables and expressions of the form $c \cdot z$, where c is an action, and z is a variable. Such insertion functions are called one-step insertions, other important classes are *head* insertion and *look-ahead* insertion functions. For head insertion the restriction on the height should not exceed 1 which refers only to the agent behavior term $G(y)$. The term $F(x)$ can be of arbitrary height. Head insertion can be reduced to one-step insertion by changing the structure of the environment but preserving the insertion equivalence of agents. In head insertion, the interaction between the environment and agent is similar to the interaction between the server and the client: a server has information only about the next step in the behavior of the client but knows everything about its own behavior. In a look-ahead insertion environment, the behavior of an agent can be analyzed for arbitrary long (but finite) future steps. We can liken such environment to the interaction between an interpreter and a program.

We consider a one-step insertion which is applied in many practical cases by restricting ourselves to purely additive insertion functions that satisfy the following conditions:

$$\left(\sum e_i\right)[u] = \sum e_i[u]$$

$$e\left[\sum u_i\right] = \sum e[u_i]$$

Given two functions $D_1 : A \times C \to 2^C$ and $D_2 : C \to 2^C$, the transition rules for insertion functions are

$$\frac{u \xrightarrow{a} u', e \xrightarrow{c} e', d \in D_1(a, c)}{e[u] \xrightarrow{d} e'[u']}$$

$$\frac{e \xrightarrow{c} e', d \in D_2(c)}{e[u] \xrightarrow{d} e'[u]}$$

We refer to D_1 and D_2 as residual functions. The first rule (interaction rule) defines the interaction between the agent and the environment which consists of choosing a matching pair of actions $a \in A$ and $c \in C$. Note that, the environment and the agent move independently. If the choice of action is made first by the environment, then the choice of action c by the environment defines a set of actions that the agent may take: a can be chosen only so that $D_1(a, c) \neq \emptyset$. The observable action d must be selected from the set $D_1(a, c)$. This selection can be restricted by the external environment if $e[u]$ considered as an agent is inserted into the environment by other agents inserted into environment $e[u]$ after u. This rule can be combined with rules for unobservable transitions if some action, say τ (as in Milner CCS), is selected in C to hide the transition. For this case we formulate the interaction rule to account for hidden interactions.

$$\frac{u \xrightarrow{a} u', e \xrightarrow{c} e', \tau \in D_1(a, c)}{e[u] \to e'[u']}$$

The second rule (environment move rule) describes the case when the environment transitions independently of the inserted agent and the agent is waiting until the environment will allow it to move. Unobservable transitions can also be combined with environment moves. Some equations should be added for the case when e or u are termination constants. We shall assume that $\bot[u] = \bot$, $0[u] = 0$, $e[\Delta] = e$, $e[\bot] = \bot$, and $e[0] = 0$. There are no specific assumptions about $\Delta[u]$ but usually neither Δ nor 0 belong to E. Note that, in the case when $\Delta \in E$ and $\Delta[u] = u$, insertion equivalence coincides with bisimulation. The definition of the insertion function for one-step insertion discussed earlier will be complete, if we assume that there are no transitions other than those defined by the rules.

The definition above can be expressed in the form of rewriting rules as follows:

$$d \in D_1(a, c) \;\Rightarrow\; (c \cdot x)[a \cdot y] \to d \cdot y$$

$$d \in D_2(c) \;\Rightarrow\; (c \cdot x)[y] \to d \cdot x[y]$$

and in the form of explicit recursive definition as

$$e[u] = \sum_{\substack{e \xrightarrow{c} e' \\ u \xrightarrow{a} u' \\ d \in D_1(a,c)}} d \cdot e'[u'] + \sum_{\substack{e \xrightarrow{c} e' \\ d \in D_2(c)}} d \cdot e'[u] + \varepsilon_e[u]$$

To compute transitions for the multiagent environment $e[u_1, u_2, \ldots, u_n]$ we recursively compute transitions for $e[u_1]$, then for $e[u_1, u_2] = (e[u_1])[u_2]$, and eventually for $e[u_1, u_2, \ldots, u_n] = (e[u_1, u_2, \ldots, u_{n-1}])[u_n]$.

Important special cases of one-step insertion functions are parallel and sequential insertion. An insertion function is called a parallel insertion if

$$e[u, v] = e[u \parallel v]$$

This means that the subsequent insertion of two agents can be replaced by the insertion of their parallel composition. The simplest example of a parallel insertion is defined as $e[u] = e \parallel u$. This special case holds when the sets of actions of environment and agents are the same ($A = C$), $b = D_1(a, a \times b)$, and $D_2(a) = A$. In the case when $\Delta \in E$, this environment is a set of all other agents interacting with a given agent in parallel, and insertion equivalence coincides with bisimilarity. Sequential insertion is introduced in a similar way:

$$e[u, v] = e[u; v]$$

This situation holds, for example, when $D_1(a, c) = \emptyset$, $D_2(c) = C$, and $\Delta[u] = u$.

6.2.3.2 Example: Agents Over a Shared and Distributed Store

As an example, consider a store, which generalizes the notions of memory, data bases, and other information environments used by programs and agents to hold data. An abstract store environment E is an environment over an action algebra C, which contains a set of actions A used by agents inserted into this environment. We shall distinguish between local and shared store environments. The former can interact with an agent inserted into it while this agent is not in a final state and, if another agent is inserted into this environment, the activity of the latter is suspended until the former completes its work. The shared store admits interleaving of the activity of agents inserted into it, and they can interact concurrently through this shared store.

6.2.3.2.1 Local and Shared Store

The residual functions for a local store are defined as:

$$D_1(a, c) = \{d | c = a \times d\}, \text{ where } d \neq \emptyset \text{ for } d \in C \backslash A \text{ or } d = \delta \text{ otherwise, and } D_2(c) = C$$

and for a shared store as

$$D_1(a, c) = \{d | c = a \times d\}, \quad \text{where } d \neq \emptyset, \ \ d \in C, \ \text{ and } \ D_2(c) = C.$$

It can be proved that the one-step insertion function for a local store is a sequential insertion and that one-step insertion for a shared store is a parallel insertion. In other words,

$$e[u_1, u_2, \ldots] = e[u_1; u_2; \ldots]$$

for a local store, and

$$e[u_1, u_2, \ldots] = e[u_1 \parallel u_2 \parallel \ldots]$$

for a shared store. The interaction move for the local store is defined as

$$\frac{u \xrightarrow{a} u', e \xrightarrow{a \times d} e'}{e[u] \xrightarrow{d} e'[u']}$$

When the store moves according to this rule, an agent inserted into it plays the role of control for this store. A store in a state $e[u]$ can only perform actions which are allowed by the agent u. This action can be combined only with an action d which is not from the action set A and cannot be used by another agent in a transition. The actions returned by the residual function are external actions and can be observed and used only from outside the store environment.

Different from a local store, in a shared store environment several agents can perform their actions in parallel according to the rule

$$\frac{u_1 \xrightarrow{a_1} u_1', \ldots, u_n \xrightarrow{a_n} u_n', e \xrightarrow{a_1 \times \cdots \times a_n \times d} e'}{e[u_1 \parallel \cdots \parallel u_n \parallel v] \xrightarrow{d} e'[u_1' \parallel \cdots \parallel u_n' \parallel v]}$$

An important special case of the store environment E is a memory over a set of names R and a data domain D. The memory can be represented by an attributed transition system with attributes R and states $e : R \to R'$. Agent actions are assignments and conditions, and their combinations are possible if they can be performed simultaneously. If a is a set of assignments, then in a transition $e \xrightarrow{a} e'$ the state e' results from applying a to e. The conjunction of conditions c enables a transition $e \xrightarrow{c \times a} e'$ if c is valid on e and $e \xrightarrow{a} e'$.

6.2.3.2.2 Multilevel Store

For a shared memory store the residual action d in the transition

$$e[u_1 \parallel \cdots \parallel u_n \parallel v] \xrightarrow{d} e'[u_1' \parallel \cdots \parallel u_n' \parallel v]$$

is intended to be used by external agents inserted later, but in a multilevel store it is convenient to restrict the interaction with the environment to a given set of agents which have already been inserted. For this purpose, a shared memory can be inserted into a higher level *closure environment* with an insertion function defined by the equation

$$g[e[u]][v] = g[e[u \parallel v]]$$

where g is a state of this environment, e is a shared memory environment, and only the following two rules are used for transitions in the closure environment:

$$\frac{e[u] \xrightarrow{c} e'[u'], c \in C_{\text{ext}} \wedge c \neq \delta}{g[e[u]] \xrightarrow{\varphi_{\text{ext}}(c,e)} g[e'[u']]}$$

$$\frac{e[u] \xrightarrow{\delta} e'[u']}{g[e[u]] \to g[e'[u']]}$$

Here C_{ext} is a distinguished set of external actions. Some of external actions can contain occurrences of names from e. The function φ_{ext} substitutes the values of these names in c and performs other transformations to make an action be observable for external environment.

Two level insertions can be described in the following way. Let $R = R_1 \cup R_2$ be divided into two nonintersecting parts: the external and internal memory. Let A_1 be the set of actions which change only the values of R_1, but can use the values of R_2 (external output actions), let A_2 be the set of actions which change only the values of R_2, but can use the values of R_1 (external input actions), and A_3 be the set of actions which change and use only the values of R_2 (internal actions). These sets are assumed to be defined on the syntactical level. Redefine the residual function D_1 and transitions of E: let $a \in A$ and split a into a combination of actions $\varphi_1(a) \times \varphi_2(a) \times \varphi_3(a)$ so that $\varphi_1(a) \in A_1$, $\varphi_2(a) \in A_2$, and $\varphi_3(a) \in A_3$

(some of these actions may be equal to δ). Define the interaction rule in the following way:

$$\frac{u \xrightarrow{a} u', e \xrightarrow{(\varphi_2(a)\sigma)\times\varphi_3(a)} e'}{e[u] \xrightarrow{c_\sigma \times \varphi_1(a)} e'[u']}$$

where σ is an arbitrary substitution of names used in conditions and in the right-hand sides of assignments of $\varphi_2(a)$ into the set of their values, $b\sigma$ is an application of the substitution σ to b, c_σ is a substitution written in the form of the condition $r_1 = \sigma(r_1) \wedge r_2 = \sigma(r_2) \wedge \cdots$. Define $\varphi_{\text{ext}}(b, e) = be$, that is a substitution of the values of R_2 to b.

Consider a two level structure of a store state

$$t[g[e_1[u_1]] \parallel g[e_1[u_1]] \parallel \cdots]$$

where $t \in D^{R_1}$ is a shared store and $e_1, e_2, \ldots \in D^{R_2}$ represent the distributed store (memory). When the component $g[e_i[u_i]]$ performs internal actions these are hidden and do not affect the shared memory. Performing external output actions change the names of the shared memory and external input actions receive values from the shared memory to change components of the distributed memory. This construction is easily iterated as the components of a distributed memory can have multilevel structure.

6.2.3.2.3 Message Passing

Distributed components can interact via shared memory. We now introduce direct interaction via message passing. Synchronous communication can be organized by extending the set of actions with a combination of actions in parallel composition independently of the insertion function. To describe synchronous data exchange in the most general abstract schema, let

$$u = \sum_{d\in D} a(d) \cdot F(d) \quad u' = \sum_{d'\in D} a'(d') \cdot F'(d')$$

be two agents which use data domain D for the exchange of information. Functions a and a' map the data domain onto the action algebra, functions F and F' map elements of the data domain onto behaviors. The parallel composition of u and u' is

$$u \parallel u' = \sum_{a(d)\times a'(d')\neq\emptyset} a(d) \times a'(d')(F(d) \parallel F'(d')) + \sum_{d\in D} a(d)(F(d) \parallel u') + \sum_{d'\in D} a'(d')(F'(d') \parallel u)$$

(note that, $\varepsilon_u = \varepsilon_{u'} = 0$, i.e., this is a special case of parallel composition where there are no termination constants). The first summand corresponds to the interaction of two agents. The other two summands reflect the possibility of interleaving. The interaction can be deterministic even if u and u' are nondeterministic if $a(d) \times a'(d') \neq \emptyset$ has only one solution. Interleaving makes it possible to select other action if $u \parallel u'$ is embedded into another parallel composition. They can also be hidden by a closure environment (similar to restriction in Calculus of Concurrent Systems, CCS).

The exchange of information through combination is bidirectional. An important special case of information exchange is the use of send/receive pairs. For example, consider the following combination rule

$$\mathbf{send}(\mathbf{addr}, d)\times\mathbf{receive}(\mathbf{addr}', d') = \begin{cases} \mathbf{exch}(\text{addr}), & \text{if addr=addr}', d=d' \\ \emptyset, & \text{otherwise} \end{cases}$$

In the latter case, if

$$u = \mathbf{send}(\mathbf{addr}, d) \cdot v$$

and

$$u' = \sum_{d' \in D} \mathbf{receive}(\mathbf{addr}, d') \cdot F(d')$$

the interaction summand of the parallel composition will be $\mathbf{exch}(\mathbf{addr}) \cdot (v \parallel F(d))$.

Asynchronous message passing via channels can be described by introducing a special communication environment. The attributes of this environment are channels and their values are sequences (queues) of stored messages. It is organized similarly to the memory environment but queue operations are used instead of storing. In addition, send and receive actions are separated in time. This environment is a special case of a store environment and can be combined with a store environment keeping separate the different types of attributes and actions.

6.2.4 Classical Theories of Concurrency

The theory of interaction of agents and environments [9–11] focuses on the description of multi-agent systems comprised of agents cooperatively working within a distributed information environment.

Other mathematical models for specifications of dynamic and real time systems interacting with environments have been developed based on process algebras (CSP, CCS, ACP, etc.), automata models (timed Büchi and Muller automata, abstract state machines [ASM]), and temporal logic (LPTL, LTL, CTL, CTL*). New models are being developed to support different peculiarities of application areas, such as Milner's π-calculus [12] for mobility and its recent extension to object-oriented descriptions.

The environment may change the predefined behavior of an agent. For example, it may contain some other agents designed independently and intended to interact and communicate with the agent during its execution. The classical theories of communication consider this interaction as part of the parallel composition of agents. The influence of the environment can be expressed as an explicit language operation such as restriction (CCS) or hiding (CSP).

In contrast to the classical theories of interaction which are based on an implicit and hence not formalized notion of an environment, the theory of interaction of agents and environments studies them as objects of different types. In our approach the environment is considered as a semantic notion and is not explicitly included in the agent. Instead, the meaning of an agent is defined as a transformation of an environment which corresponds to inserting the agent into its environment. When the agent is inserted into the environment, the environment changes and this change is considered to be a property of the agent described.

6.2.4.1 Process Algebras

An algebraic theory of concurrency and communication that deals with the occurrence of events rather than with updates of stored values is called a process algebra. The main variants of process algebra are generally known by their acronyms: CCS [8] — Calculus of Concurrent Systems developed by Milner, CSP [13] — Hoare's Communicating Sequential Processes, and ACP — Algebra of Communicating Processes of Bergstra and Klop [14]. These theories are based on transition systems and bisimulation, and consider interaction of composed agents. They employ nondeterministic choice as well as parallel and sequential compositions as primitive constructs. The influence of the environment on the system may be expressed as an explicit language operation, such as restriction in CCS or hiding in CSP. These theories consider communicating agents as objects of the same type (this type may be parameterized by the alphabets for events or actions) and define operations on these types.

The CCS model specifies sets of states of systems (processes) and transitions between these states. The states of a process are terms and the transitions are defined by the operational semantics of the computation, which indicates how and under which conditions a term transforms itself into another term. Processes are represented by the synchronization tree (or process graph). Two processes are identified through bisimulation.

CCS introduces a special action τ, called the silent action, which represents an internal and invisible transition within a process. Other actions are split into two classes: *output* actions, which are indicated by an overbar, and *input* actions, which are not decorated. Synchronization only takes place between a single input and a single output, and the result is always the silent action τ. Thus, $a \times \tilde{a} = \tau$, for all actions a. Consequentially, communication serves only as synchronization; its result is not visible.

The π-calculus [12] is an enhancement of CCS and models concurrent computation by processes that exchange messages over named channels. A distributed interpretation of the π-calculus provides for synchronous message passing and nondeterministic choice. The π-calculus focuses on the specification of the behavior of mobile concurrent processes, where "mobility" refers to variable communication via named channels, which are the main entities in the π-calculus. Synchronization takes place only between two channel agents when they are available for interchange (a named output channel is indicated by an overbar, while an input channel with the same name is not decorated). The influence of the environment in the π-calculus is expressed as an explicit operation of the language (hiding). As a result of this operation, a channel is declared inaccessible to the environment.

CSP explicitly differentiates the set of atomic actions that are allowed in each of the parallel processes. The parallel combinator is indexed by these sets: when $(P_{\{A\}} \parallel Q_{\{B\}})$, P engages only in events from the set A, and Q only in events from the set B. Each event in the intersection of A and B requires a synchronous participation of both processes, whereas other events only require participation of the relevant single process. As a result, $a \times \tilde{a} = a$, for all actions a. The associative and commutative binary operator $\times$ describes how the output data supplied by two processes is combined before transmission to their common environment.

In CSP, a process is considered to run in an environment which can veto the performance of certain atomic actions. If, at some moment during the execution, no action, in which the process is prepared to engage in, is allowed by the environment, then a deadlock occurs, which is considered to be observable. Since in CSP a process is fully determined by the observations obtainable from all possible finite interactions, a process is represented by its failure set. To define the meaning of a CSP program, we determine the set of states corresponding to normal termination of the program, and the set of states corresponding to its failures. Thus, the CSP semantics is presented in model-theoretic terms: two CSP processes are identified if they have the same failure set (failure equivalence).

The main operations of ACP are prefixing and nondeterministic choice. This algebra allows an event to occur with the participation of only a subset of the concurrently active processes perhaps omitting any that are not ready. As a result, the parallel composition of processes is a mixture of synchronization and interleaving, where each of the processes either occurs independently or is combined by $\times$ with a corresponding event of another process. The merge operator is defined as

$$\text{Merge}(a, b) = (a \times b) + (a; b) + (b; a).$$

ACP defines its semantics algebraically; processes are identified through bisimulation.

Most differences between CCS, ACP, and CSP can be attributed to differences in the chosen style of presentation of the semantics: the CSP theory provides a model, illustrated with algebraic laws. CCS is a calculus, but the rules and axioms in this calculus are presented as laws, valid in a given model. ACP is a calculus that forms the core of a family of axiomatic systems, each describing some features of concurrency.

6.2.4.2 Temporal Logic

Temporal logic is a formal specification language for the description of various properties of systems. A temporal logic is a logic augmented with temporal modalities to allow a specification of the order of events in time, without introducing time explicitly as a concept. Whereas traditional logics can specify properties relating to the initial and final states of terminating systems, a temporal logic is better suited to describe the on-going behavior of nonterminating and interacting (reactive) systems.

As an example, Lamport's TLA (Temporal Logic of Actions) [5,15] is based on Pnueli's temporal logic [16] with assignment and an enriched signature. It supports syntactic elements taken from programming

languages to ease maintenance of large-sized specifications. TLA uses formulae on *behavior*, which are considered as a sequence of *states*. States in TLA are assignments of values to variables. A system satisfies a formula iff that formula is true in all behaviors of this system. Formulae where the arguments are only the old and the new states are called *actions*.

Here, we distinguish between *linear* and *branching* temporal logics. In a linear temporal logic, each moment of time has a unique possible future, while in branching temporal logic, each moment of time may have several possible futures. On one hand, linear temporal logic formulae are interpreted over linear sequences of points in time and specify the behavior of a single computation of a system. Formulae of a branching temporal logic, on the other hand, are interpreted over tree-like structures, each describing the behavior of possible computations of a nondeterministic system.

Many temporal logics are decidable and corresponding decision procedures exist for linear and branching time logics [17], propositional modal logic [18], and some variants of CTL* [19]. These decision procedures proceed by building a canonical model for a set of temporal formulae representing properties of the system to be verified by using techniques from automata theory, semantic tableaux, or binary decision diagrams [20]. Determining whether such properties hold for a system amounts to establishing that the corresponding formulae are true in a model of the system. *Model checking* based on these decision procedures has been successfully applied to find subtle errors in industrial-size specifications of sequential circuits, communication protocols, and digital controllers [21].

Typically, a system to be verified is modeled as a (finite) state transition graph, and their properties are formulated in an appropriate propositional temporal logic. An efficient search procedure is then used to determine whether the state transition graph satisfies the temporal formulae or not. This technique was first developed in the 1980s by Clarke and Emerson [22] and by Quielle and Sifakis [23] and extended later by Burch et al. [21].

Examples of temporal properties (properties of the interaction between processes in a reactive system) are as diverse as their applications (this classification was introduced in References 2 and 24):

- *Safety properties* state that "something bad never happens" (a program never enters an unacceptable state).
- *Liveness properties* state that "something good will eventually happen" (a program eventually enters a desirable state).
- *Guarantees* specify that an event will eventually happen but does not promise repetitions.
- *Obligations* are disjunctions of safety and guarantee formulae.
- *Responses* specify that an event will happen infinitely many times.
- *Persistence* specifies the eventual stabilization of a system condition after an arbitrary delay.
- *Reactivity* is the maximal class formed from the disjunction of response and persistence properties.
- *Unconditional Fairness* states that a property p holds infinitely often.
- *Weak Fairness* states that if a property p is continuously true then the property q must be true infinitely often.
- *Strong Fairness* states that if a property p is true infinitely often then the property q must be true infinitely often.

6.2.4.3 Timed Automata

Timed automata accept timed words — infinite sequences in which a real-valued time of occurrence is associated with each symbol. A timed automaton is a finite automaton with a finite set of real-valued clocks. The clocks can be reset to 0 (independent of each other) with the transitions of the automaton and keep track of the time elapsed since the last reset. Transitions of the automaton put certain constraints on the clock value such that a transition may be taken only if the current values of the clocks satisfy the associated constraints.

Timed automata can capture qualitative features of real time systems such as liveness, fairness, and nondeterminism, as well as its quantitative features such as periodicity, bound response, and timing delays.

ned automata are a generalization of finite ω-automata (either Büchi automata and Muller automata [25,26]). When Büchi automata are used for modeling finite-state concurrent processes, the verification problem is reduced to that of language inclusion [27,28]. While the inclusion problem for ω-regular languages is decidable [26], for timed automata the inclusion problem is undecidable, which constitutes a serious obstacle in using timed automata as a specification language for validation of finite-state real time systems [29].

6.2.4.4 Abstract State Machine

Gurevich's ASM project [30,31] attempts to apply formal models of computation to practical specification methods.

ASM assumes the "Implicit Turing Thesis" according to which every algorithm can be modeled at its appropriate abstraction level (its algorithm) by a corresponding ASM. ASM descriptions are based on the concept of evolving algebras, which are transition systems on static algebras. Each static algebra represents a state of the modeled system and transition rules are transitions in the modeled system. To simplify the semantics and ease proofs, transitions are limited: they can change only functions, but not sorts, and cannot directly change the universe.

A single agent ASM is defined over a vocabulary (a set of functions, predicates, and domains). Its states are defined by assigning an interpretation to the elements of the vocabulary. An ASM program describes the rules of transitioning between states. An ASM program is defined by basic transition rules, such as updates (changes to the interpretation of the vocabulary), conditions (apply only if some specific condition holds), or choice (extracts from a state elements with given properties), or by combinations of transition rules into complex rules.

Multiagent ASM consist of a number of agents that execute their ASM program concurrently and interact through globally shared locations of a state. Concurrency between agents is modeled by *partially ordered runs*. The program steps executed by each agent are linearly ordered; in addition, program steps in different programs are ordered if they represent causality relations. Multiagent ASM rely on a continuous global system time to model time-related aspects.

6.2.4.5 Rewriting Logic

Rewriting logic [32] allows to prove assertions about concurrent systems with states changing under transitions. Rewriting logic extends equational logic and constitutes a logical framework in which many logics and semantic formalisms can be represented naturally (i.e., without distorting encoding). Similar to algebras allowing a semantic interpretation to equational logic, models of rewriting logic are concurrent systems. Moreover, models of concurrent computation, object-oriented design languages, architectural description languages, and languages for distributed components also have natural semantics in rewriting logic [33].

In rewriting logic, system states are in a bijective correspondence with formulae (modulo whatever structural axioms are satisfied by such formulae, e.g., modulo associativity or commutativity of connectives) and concurrent computations in a system are in a bijective correspondence with proofs (modulo appropriate notions of equivalence). Given this equivalence between computation and logic, a rewriting logic axiom of the form

$$t \rightarrow t'$$

has two readings. Computationally, it means that a fragment of a system state that is an instance of the pattern t can change to the corresponding instance of t' concurrently with any other state changes. The computational meaning is that of a local concurrent transition. Logically, it just means that we can derive the formula t from the formula t', that is, the logical reading is that of an inference rule. Computation consists of rewriting to a normal form, that is, an expression that can no further be rewritten; when the normal form is unique, it is taken as the value of the initial expression. When rewriting equal terms always leads to the same normal form, the set of rules is said to be confluent and rewriting can be used to check for equality.

Rewriting logic is reflective [34,35], and thus, important aspects of its meta-theory can be represented at the object level in a consistent way. The language Maude [36,37] has been developed at SRI to implement a framework for rewriting logic. The language design and implementation of Maude systematically leverage the reflexivity of rewriting logic and make the meta-theory of rewriting logic accessible to the user allowing to create within the logic a formal environment for the logic with tools for formal analysis, transformation, and theorem proving.

6.3 Requirements Capture and Validation

In Reference 38, requirements capture is defined as an engineering process for determining what artifacts are to be produced as the result of the development effort. The process involves the following steps:

- Requirements identification
- Requirements analysis
- Requirements representation
- Requirements communication
- Development of acceptance criteria and procedures

Requirements can be considered as an agreement between the customer and the developer. As agreements, they must be understandable to the customer as well as to the developer and the level of formalization depends on the common understanding and the previous experience of those involved in the process of requirements identification.

The main properties of the system to be developed including its purpose, functionality, conditions of use, efficiency, safety, liveness, or fairness properties, are specified in the requirements along with the main goals of the development project. The requirements also include an explanation of terms referenced, information about other already developed systems which should be reused in the development process, and possible decisions on implementation and structuring of the system.

It is well recognized that identifying and correcting problems in requirements and early design phase avoids far more expensive repairs later. Boehm quotes late life-cycle fixes to be a hundred times more expensive than corrections made during the early phases of system development [39]. In Reference 40, Boehm documents that the relative cost of repairing errors increases exponentially with the life-cycle phase at which the error was detected. Kelly et al. [41] documents a significantly higher density of defects found during the requirements phase as compared with later life-cycle phases. Early life-cycle defects are also very prevalent: in Reference 42, it was shown that of 197 critical faults found during integration testing of spacecraft software, only 3 were programming mistakes. The other 194 were introduced at earlier stages. Fifty percent of these faults were owing to flawed requirements (mainly omissions) for individual components, 25% were owing to flawed designs for these components, and the remaining 25% were owing to flawed interfaces between components and incorrect interactions between them. A number of other studies [43,44] reveal that most errors in software systems originate in the requirements and functional specifications. If the errors were detected as soon as possible, their repair would have been least expensive.

A requirements specification must describe the external *behavior* of a system in terms of observable events and actions. The latter describes interaction of system components with their environments including other components and acceptable parts of the external physical world, that is, those aspects of the world which can influence the behavior of the system.

We consider requirements to be correct if they are consistent and complete. Checking consistency and completeness of requirements is the final task of requirements representation.

The informal understanding of the consistency of requirements is the existence of an implementation which satisfies the requirements. In other words, if requirements are inconsistent, then an implementation free of errors (bugs) that satisfies all the requirements cannot be created.

Unfortunately, most ways for describing requirements and preliminary designs (natural language, diagrams, pseudocode) do not offer mechanized means of establishing correctness, so the primary means

to deduce their properties and consequences is through inspections and reviews. To be amenable to analysis, the requirements must first be formalized, that is, rewritten in some formal language. We call this formalized description a *requirements specification*. It should be free of implementation details and can be used for the development of an executable model of a system, which is called an *executable specification*. Note that, any description that is formal enough to be amenable to operational interpretation will also provide some method of implementation, albeit usually a rather inefficient one.

The existence of an executable specification which satisfies the formalized requirements is a sufficient condition of consistency. It is sufficient, because a final implementation that is free of errors can be extracted from executable specifications using formal methods such as stepwise refinement.

Completeness of requirements is understood as the uniqueness of the executable specification considered up to some equivalence. The intuitive understanding of equivalence is as follows: two executable specifications are equivalent if they demonstrate the same behaviors in the same environments. Completeness can also be expressed in terms of determinism of the executable specification (requirements are sufficient for constructing a unique deterministic model). In some cases, incompleteness of requirements is not harmful because it can be motivated by the necessity to suspend implementation decisions until later stages of development.

The correspondence between the original requirements and the requirements specification is not formal. Experience has shown that special skills are required to check correspondence between informal and formal requirements. Incompleteness and inconsistencies discovered at this stage are used for improvement and correction of the requirements which are used to correct the requirements specification.

6.3.1 Approaches to Requirements Validation

Standard approach to requirements analysis and validation typically involve manual processes such as "walk-throughs" or Fagan-style inspections [45,46]. The term walk-through refers to a range of activities that can vary from cursory peer reviews to formal inspections, although walk-throughs usually do not involve the replicable processes and methodical data collection that characterize Fagan-style inspections. Fagan's highly structured inspection process was originally developed for hardware logic and later applied to software design and code and eventually extended to all life-cycle phases, including requirements development and high level design [45].

A Fagan inspection involves a review team with the following roles: a *Moderator*, an *Author*, a *Reader*, and a *Tester*. The Reader presents the design or code to the other team members, systematically walking through every piece of logic and every branch at least once. The Author represents the viewpoint of the designer or coder, and the test perspective is represented by the Tester. The Moderator is trained to facilitate intensive but constructive discussion. When the functionality of the system is well understood, the focus shifts to a search for faults, possibly using a checklist of likely errors to guide the process. The inspection process includes highly structured rework. One of the main advantages of Fagan-style inspections over other conventional forms of verification and validation is that inspections can be applied early in the life cycle. Thus potential anomalies can be detected before they become entrenched in low level design and implementation.

Rushby [47] gives an overview of techniques of mechanized formal methods: decision procedures for specialized, but ubiquitous, theories such as arithmetic, equality, and the propositional calculus are helpful in discovering false theorems (especially if they can be extended to provide counter examples) as well as in proving true ones, and their automation dramatically improves the efficiency of proof. Rewriting is essential to efficient mechanization of formal methods. Unrestricted rewriting provides a decision procedure for theories axiomatized by terminating and confluent sets of rewrite rules, but few such theories arise in practice.

Integration of various techniques increases the efficiency of these methods. For example, theorem proving attempts to show that a formula follows from given premises, while model checking attempts to show that a given system description is a model for the formula. An advantage of model checking is that, for certain finite-state systems and temporal logic formulae, it can be automated and is more efficient than

theorem proving. Benefits of an integrated system (providing both theorem proving and model checking) are that model checking can be used to discharge some cases in a larger proof and theorem proving can be used to justify reduction to a finite state that is required for automated model checking. Integration of these techniques can provide a further benefit: before undertaking a potentially difficult and costly proof, we may be able to use model checking to examine special cases. Any errors that can be discovered and eliminated in this way will save time and effort during theorem proving.

Model checking determines whether a given formula stating a property of the specification is satisfied in a Kripke-model (in the specification represented as a Kripke-model). In the worst case, these algorithms must traverse the whole of the model, that is, visit all states of the corresponding transition system, and consequentially, model checking can be applied mainly for finite-state systems even in the presence of sophisticated means of representing the set of states, such as binary decision diagram (BDD) methods [20], while the proof of theorems about infinite state systems can be done only by means of deductive methods.

6.3.2 Tools for Requirements Validation

Formal methods may be classified according to their primary purpose as descriptive or analytic. Descriptive methods focus largely on specifications as a medium for review and discussion, whereas analytic methods focus on the utility of specification as a mathematical model for analyzing and predicting the behavior of systems. Not surprisingly, the different emphasis is reflected in the type of a formal language favored by either methods.

Descriptive formal methods emphasize the expressive power of the underlying language and provide a rich type system, often leveraging the notations of conventional mathematics or set theory. These choices in language elements do not readily support automation; instead, descriptive methods typically offer attractive user interfaces and little in the way of deductive machinery. These methods assume that the specification process itself serves as verification, as expressing the requirements in mathematical form leads to detect inconsistencies that are typically overlooked in natural language descriptions. Examples of primarily descriptive formal methods are VDM [48], Z [49], B [50], or LOTOS [51].

Analytic formal methods place emphasis on mechanization and favor specification languages that are less expressive but capable of supporting efficient automated deduction. These methods vary in the degree of automation provided by the theorem prover, or, conversely, by the amount of user interaction in the proof process. They range from automatic theorem proving without user interaction to proof checking without automatic proof steps. The former typically have restricted specification languages and powerful provers that can be difficult to control and offer little feedback on failed proofs, but perform impressively in the hands of experts, for example, Nqthm [52]. Proof checkers generally offer more expressive languages, but require significant manual input for theorem proving, for example, high-order logic (HOL) [53]. Many tools fall somewhere in between, depending on language characteristics and proof methodology, for example, Eves [54,55], or PVS [56]. The goal of mechanized analysis may be either to prove the equivalence between different representations of initial requirements or to establish properties that are considered critical for correct system behavior (safety, liveness, etc.).

Tools for analytic formal methods fall into two main categories: state exploration tools (model checkers) and deductive verifiers (automated theorem provers). Model checking [57] is an approach for formally verifying finite-state systems. Formalized requirements are expressed as temporal logic formulas, and efficient symbolic algorithms are used to process a model of the system and check if the specification holds in that model. Widely known tools are VeriSoft [58], SMV [59], or SPIN [60]. Some deductive verifiers either support inference in first order (Larch [61]), others, such as PVS [62], are based on higher-order languages integrated with supporting tools and interactive theorem provers.

Today, descriptive methods are often augmented by facilities of mechanized analysis. Also, automated theorem proving and model checking approaches may be integrated in a single environment. For example, a BDD-based model checker can be used as a decision procedure in the PVS [63] theorem prover. In addition to the prover or proof checker, a key feature of analytic tools is the type checker which checks

specifications for semantic consistency, possibly adding semantic information to the internal representation built by the parser. If the type system of the specification language is not decidable, theorem proving may be required to establish the type consistency of a specification. An overview of verification tools and underlying methods can be found in Reference 64.

The most severe limitation to the deployment of formal methods is the need for mathematical sophisticated users, for example, with respect to the logical notation these methods use, which is often an insurmountable obstacle to the adoption of these methods in engineering disciplines. General-purpose decision procedures, as implemented in most of these methods, are not scalable to the size of industrial projects. Another obstacle to the application of deductive tools like PVS is the necessity to develop a mathematical theory formalized on a very detailed level to implement even very simple predicates. Recently, formal methods were successfully applied to specification and design languages widely accepted in the engineering community, such as MSC [65], SDL [51], or UML [66]. Several tool vendors participate in the OMEGA project aimed at the development of formal tools for the analysis and verification of design steps based on UML specifications [67]. SDL specifications can be checked by model checkers as well as automated theorem provers: for example, the IF system from Verimag converts SDL to PROMELA as input to the SPIN model checker [68]. At Siemens, verification of the GSM protocol stack was conducted using the BDD-based model checker SVE [69]. An integrated framework for processing SDL specification has been implemented based on the automated theorem prover ACL2 [70]. Ptk [71] provides semantic analysis of MSC diagrams and generates test scripts from such diagrams in a number of different languages including SDL, TTCN, and C. FatCat [72] locates situations of nondeterminacy in a set of MSC diagrams.

In the following, we give a cursory overview of some of the wider known tools supporting the application of formal methods to specifications. This survey reviews only tools that are freely available, at least for research use. A number of powerful commercial verification technologies have been developed, for example: ACL2 (Computational Logic, USA), ASCE (Adelard, UK), Atelier B (STERIA Méditerranée, France), B-Toolkit (B-Core, UK), CADENCE (Cadence, USA), Escher (Escher Technologies, UK), FDR (Formal Systems, UK), ProofPower (ICL, UK), Prover (Prover Technology, Sweden), TAU (Telelogic), Valiosys (Valiosys, France), and Zola (Imperial Software, UK). A more detailed survey of commercial tools aimed at formal verification is given in Reference 73.

The tools and environments surveyed provide only a sample of the wide variety of tools available. In particular, in the area of model checking a large number of implementations support the verification of specifications written in different notations and supporting different temporal logics. For example, Kronos [74] allows modeling of real time systems by timed automata, that is, automata extended with a finite set of real-valued clocks, used to express timing constraints. It supports TCTL, an extension of temporal logic that allows quantitative temporal claims. UPAAL [75] represents systems as networks of automata extended with clocks and data variables. These networks are compiled from a nondeterministic guarded command language with data types. The VeriSoft [58] model checker explores the state space of systems composed of concurrent processes executing arbitrary code (written in any language) and searches for concurrency pathologies such as deadlock, lifelock, divergence, and for violation of user-specified assertions.

6.3.2.1 Descriptive Tools

6.3.2.1.1 Vienna Development Method [76]

Vienna development method (VDM) is a model-oriented formal specification and design method based on discrete mathematics, originally developed at IBM's Vienna Laboratory. It is a model-oriented formal specification and design method based on discrete mathematics. Tools to support formalization using VDM include parsers, typecheckers, proof support, animation, and test case generators. In VDM, a system is developed by first specifying it formally and proving that the specification is consistent, then iteratively refining and decomposing the specification provided that each refinement satisfies the previous specification. This process continues until the implementation level is reached.

The VDM Specification Language (VDM-SL) was standardized by ISO in 1996 and is based on first-order logic with abstract data types. Specifications are written as constructive specifications of an abstract

data type, by defining a class of objects and a set of operations that act upon these objects. The model of a system or subsystem is then based on such an abstract data type. A number of primitive data types are provided in the language along with facilities for user-defined types.

VDM has been used extensively in Europe [48,77,78]. Tools have been developed to support formalization using VDM, for example, Mural [79] which aids formal reasoning via a proof assistant.

6.3.2.1.2 Z [49,80]

Z evolved from a loose notation for formal specifications to a standardized language with tool support provided by a variety of third parties. The formal specification notation has been developed by the Programming Research Group at Oxford University. It is based on Zermelo-Fraenkel set theory and first-order predicate logic. Z is supported by graphical representations, parsers, typecheckers, pretty-printers, and a proof assistant implemented in HOL providing proof checkers as well as a full-fledged theorem prover. The standardization of Z through ISO solidified the tool base and enhanced interest in mechanized support.

The basic Z form is called a schema, which is used to introduce functions. Models are constructed by specifying a series of schemata using a state transition style. Several object-oriented extensions to Z have been proposed.

Z has been used extensively in Europe (primarily in the United Kingdom) to write formal specifications for various industrial software development efforts and has resulted in two awards for technological achievement: for the IBM CICS project and for a specification of the IEEE standard for floating-point arithmetic. To leverage Z in embedded systems design, Reference 81 extended Z by temporal interval logic and automated reasoning support through Isabelle and the SMV model checker.

6.3.2.1.3 B [50]

Following the B method, initial requirements are represented as a set of abstract machines, for which an object-based approach is employed at all stages of development. B relies on a wide-spectrum notation to represent all levels of description, from specification through design to implementation.

After specifying requirements, they can be checked for consistency which for B means preservation of invariants. In addition, B supports checking correctness of the refinement steps to design and implementation.

B is supported through toolkits providing syntax checkers, type checkers, a specification animator, proof-obligation generator, provers allowing different degrees of mechanization, and a rich set of coding tools. It also includes convenient facilities for documenting, cross-referencing, and reviewing specifications.

The B method is popular in industry as well as in the academic community. Several international conferences on B have been conducted. An example of the use of B in embedded systems design is reported in Reference 82 which promotes the development of correct software for smart cards through translation of B specifications into embedded C code.

6.3.2.1.4 Rigorous Approach to Industrial Software Engineering [83,84]

Rigorous Approach to Industrial Software Engineering (RAISE) is based on a development methodology that evolved from the VDM approach. Under the RAISE methodology, development steps are carefully organized and formally annotated using the RAISE specification language which is a powerful wide-spectrum language for specifying operations and processes allowing derivations between levels of specifications. It provides different styles of specification: model-oriented, algebraic, functional, imperative, and concurrent. The CORE requirements method is also provided as an approach for front-end analysis. Supporting tools provide a window-based editor, parser, typechecker, proof tools, a database, and translators to C and Ada.

Derivations from one level to the next generate proof obligations. These obligations may be discharged using proof tools which are also used to perform validation (establishing system properties). Detailed descriptions of the development steps and overall process are available for each tool. The final implementation step has been partially mechanized for common implementation languages.

6.3.2.1.5 Common Algebraic Specification Language [85]

Common Algebraic Specification Language (CASL) was developed as a language for formal specification of functional requirements and modular software design that subsumes many algebraic specification frameworks and also provides tool interoperability. CASL is a complex language with a complete formal semantics comprising a family of formally defined specification languages meant to constitute a common framework for algebraic specification and development [86]. To make tool construction manageable, it allows for reuse of existing tools, for interoperability of tools developed at different sites, and for construction of generic tools that can be used for several languages.

The CASL Tool Set, CATS, combines a parser, a static checker, a pretty printer, and facilities for translation of CASL to a number of different theorem provers. Encoding eliminates subsorting and partiality, and thus allows reuse of existing theorem proving tools and term rewriting engines for CASL. Typical applications of a theorem prover in the context of CASL are checking semantic correctness (according to the model semantics) by discarding proof obligations that have been generated during checking of static semantic constraints and validating intended consequences, which can be added to a specification using annotations. This allows a check for consistency with informal requirements.

In the scope of embedded systems verification, the Universal Formal Methods (UniForM) Workbench has been deployed in the development of railway control and space systems [87]. This system aims to provide a basis for interoperability of tools and the combination of languages, logics, and methodologies. It supports verification of basic CASL specification encoded in Isabelle and the subsequent implementation of transformation rules for CASL to support correct development by transformation.

6.3.2.1.6 Software Cost Reduction [88]

Software cost reduction (SCR) is a formal method for modeling and validating system requirements. SCR models a system as a black box computing output data from the input data. System behavior is represented as a finite-state machine. SCR is based on tabular notations that are relatively easy to understand. To develop correct requirements with SCR, a user shall perform four types of activities supported by SCR tools.

First, a specification is developed using the SCR tabular notation using the specification editor. Second, the specification is automatically analyzed for violations of application-independent proprieties, such as nondeterminism and missing cases, using an extension of the semantic tableaux algorithm. To validate the specification, the user may run scenarios, sequences of observable events, through the SCR simulator and for checking application-dependent properties the user can apply the Spin model checker by translating the specification into Promela.

A toolset has been developed by the Naval Research Laboratory, including a specification editor, a simulator for symbolically executing the specification, and formal analysis tools for testing the specification for selected properties.

SCR has been applied primarily to the development of embedded control systems including the A-7E aircrafts operational flight program, a submarine communications system, and safety-critical components of two nuclear power plants [89].

6.3.2.1.7 EVES [55]

EVES is an integrated environment supporting formal development of systems from requirements to code. Additionally, it may be used for formal modeling and mathematical analysis. To date, EVES applications have primarily been in the realm of security-critical systems.

EVES relies on the wide-spectrum language Verdi, ranging from a variant of classical set theory with a library mechanism for information hiding and abstraction to an imperative programming language. The EVES mathematics is based on ZFC set theory without the conventional distinction between terms and formulae. Supporting tools are a well-formedness checker, the integrated automated deduction system NEVER, a proof checker, reusable library framework, interpreter, and compiler.

Development is treated as theory extension: each declaration extends the current theory with a set of symbols and axioms pertaining to those symbols. Proof obligations are associated with every declaration

to guarantee conservative extension. The EVES library is a repository of reusable concepts (e.g., a variant of the Z mathematical toolkit is included with EVES) and is the main support for scaling, information hiding, and abstraction. Library units are either specification units (axiomatic descriptions), model units (models or implementations of specifications), or freeze units (for saving work in progress).

6.3.2.2 Deductive Verifiers

6.3.2.2.1 High-Order Logic [53]

High-order logic is an environment for interactive theorem proving in HOL, that is, predicate calculus with terms from the typed lambda calculus. HOL provides a parser, pretty-printer, typechecker, as well as forward and goal oriented theorem provers. It interfaces HOL to the MetaLanguage (ML) which allows representation of terms and theorems of the logic, of proof strategies, and of logical theories.

The HOL system is an interactive mechanized proof assistant. It supports both forward and backward proofs. The forward proof style applies inference rules to existing theorems in order to obtain new theorems and eventually the desired goal. Backward or goal oriented proofs start with the goal to be proven. Tactics are applied to the goal and subgoals until the goal is decomposed into simpler existing theorems.

HOL provides a general and expressive vehicle for reasoning about various classes of systems. Some of the applications of HOL include the specification and verification of compilers, microprocessors, interface units, algorithms, and formalization of process algebras, program refinement tools, and distributed algorithms.

Initially, HOL was aimed at hardware specification and verifying but later its application was extended to many other domains. Since 1988 an annual meeting of the HOL community evolved into a large international conference.

6.3.2.2.2 Isabelle [90]

Isabelle, developed at Cambridge University, is a generic theorem prover providing a high degree of automation and supporting a wide variety of built-in logics: many-sorted first-order logic, constructive and classical versions, higher-order logic, Zermelo–Fraenkel set theory, an extensional version of Martin-Löf's Type Theory, two versions of the Logic for Computable Functions, the classical first-order sequent calculus, and modal logic. New logics are introduced by specifying their syntax and inference rules. Proof procedures can be expressed using tactics. A generic simplifier performs rewriting by equality relations and handles conditional and permutative rewrite rules, performs automatic case splits, and extracts rewrite rules from context. A generic package supports classical reasoning in a first-order logic, set theory, etc. The proof process is automated to allow long chains of proof steps, reasoning with and about equations, and proofs about facts of linear arithmetic.

Isabelle aims at the formalization of mathematical proofs. Some large mathematical theories have been formally verified and are available to a user. These include elementary number theory, analysis, and set theory. For example, Isabelle's Zermelo–Fraenkel set theory derives general theories of recursive functions and data structures (including mutually recursive trees and forests, infinite lists, and infinitely branching trees).

Isabelle has been applied to formal verification as well as reasoning about the correctness of computer hardware, software, and computer protocols. Reference 91 has applied Isabelle to prove correctness of safety-critical embedded software: an HOL implemented in Isabelle has been used to model both specification and implementation of initial requirements; the problem of implementation correctness is reduced to a mathematical theorem to be proven.

6.3.2.2.3 PVS [56,62]

PVS provides an integrated environment for the development and analysis of formal specifications and is intended primarily for the formalization of requirements and design-level specifications, and for the rigorous analysis of difficult problems. It has been designed to benefit from synergetic usage of different formalisms in its unified architecture.

The PVS specification language is based on classical, typed HOL with predicate subtypes, dependent typing, and abstract data types. The highly expressive language is tightly integrated with its proof system and allows automated reasoning about type dependencies. PVS offers a rich type system, strict typechecking, and powerful automated deduction with integrated decision procedures for linear arithmetic and other useful domains, and a comprehensive support environment. PVS specifications are organized into parameterized theories that may contain assumptions, definitions, axioms, and theorems. Definitions are guaranteed to provide conservative extension. Libraries of proved specifications from a variety of domains are available.

The PVS prover supports a fully automated mode as well as an interactive mode. In the latter, the user chooses among various inference primitives (induction, quantifier reasoning, conditional rewriting, simplification using specific decision procedures, etc.). Automated proofs are based on user-defined strategies composed from inference primitives. Proofs yield scripts that can be edited and reused. Model-checking capabilities are integrated with the verification system and can be applied for automated checking of temporal properties.

PVS has been applied to algorithms and architecture for fault-tolerant flight control systems, to problems in real-time system design, and to hardware verification. Reference 92 combined PVS with industrial, UML-based development. Similarly, the Boderc project at the Embedded Systems Institute aims to integrate UML-based software design for embedded systems into a common framework that is suitable for multidisciplinary system engineering.

6.3.2.2.4 Larch [61]

Larch is a first-order specification language supporting equational theories embedded in a first-order logic. The Larch Prover (LP) is designed to treat equations as rewrite rules and carry out other inference steps such as induction and proof by cases. The user may introduce operators and assertions about operators as part of the formalization process. The system is comprised of parser, type checker, and a user-directed prover.

Larch Prover is designed to work midway between proof checking and fully automatic theorem proving. Users may direct the proof process at a fairly high level. LP attempts to carry out routine steps in a proof automatically and provide useful information about why proofs fail, but is not designed to find difficult proofs automatically.

6.3.2.2.5 Nqthm [52]

Nqthm is a toolset based on the powerful heuristic Boyer–Moore theorem prover for a restricted logic (a variant of pure applicative Lisp). There is no explicit specification language; rather, one writes specifications directly in the Lisp-like language that encodes the quantifier-free, untyped logic. Recursion is the main technique for defining functions and, consequentially, mathematical induction is the main technique for proving theorems. The system consists of parser, pretty-printer, limited typechecker (the language is largely untyped), theorem prover, and animator.

The highly automated prover can be driven by large databases of previously supplied (and proven) lemmas. The tool distribution comes with many examples of formalized and proved applications. For over a decade, the Nqthm series of provers has been used to formalize a wide variety of computing problems including safety-critical algorithms, operating systems, compilers, security devices, microprocessors, and pure mathematics. Two well-known industrial applications are a model of a Motorola digital signal processing (DSP) chip and the proof of correctness of the floating point division algorithm for the AMD5K 86 microprocessor.

6.3.2.2.6 Nuprl [93]

Nuprl was originally designed by Bates and Constable at Cornell University and has been expanded and improved over the past 15 yr by a large group of students and research associates. Nuprl is a highly extensible open system that provides for interactive creation of proofs, formulae, and terms in a typed language which is constructive type theory with extensible syntax. The Nuprl system supports HOLs and rich type theories. The logic and the proof systems are built on a highly regular untyped term structure,

a generalization of the lambda calculus and mechanisms given for reduction of such terms. The style of the Nuprl logic is based on the stepwise refinement paradigm for problem solving in which the system encourages the user to work backwards from goals to subgoals until one reaches what is known.

Nuprl provides a window-based interactive environment for editing, proof generation, and function evaluation. The system incorporates a sophisticated display mechanism that allows users to customize the display of terms. Based on structure editing, the system is free to display terms without regard to parsing of syntax. The system also includes the functional programming language ML as its metalanguage; users extend the proof system by writing their own proof generating programs (tactics) in ML. Since tactics invoke the primitive Nuprl inference rules, user extensions via tactics cannot corrupt system soundness. The system includes a library mechanism and is provided with a set of libraries supporting the basic types including integers, lists, and Booleans.

The system also provides an extensive collection of tactics. The Nuprl system has been used as a research tool to solve open problems in constructive mathematics. It has been used in formal hardware verification, as a research tool in software engineering, and to teach mathematical logic to Cornell undergraduates. It is now being used to support parts of computer algebra and is linked to the Weyl computer algebra system.

6.3.2.3 State Exploration Tools

6.3.2.3.1 Symbolic Model Verifier (SMV) [59]

The SMV system is a tool for checking finite-state systems against specifications of properties. Its high-level description language supports modular hierarchical descriptions and the definition of reusable components. Properties are described in Computation Tree Logic (CTL), a propositional, branching-time temporal logic. It covers a rich class of properties including safety, liveness, fairness, and deadlock freedom.

The SMV input language offers a set of basic data types consisting of bounded integer subranges and symbolic enumerated types, which can be used to construct static, structured types. SMV can handle both synchronous and asynchronous systems, and arbitrary safety and liveness properties. SMV uses a BDD-based symbolic algorithm of model checking to avoid explicitly enumerating the states of the model. With carefully tuned variable ordering, the BDD algorithm yields a system capable of verifying circuits with extremely large numbers of states.

The SMV system has been distributed widely and has been used to verify industrial-scale circuits and protocols, including the cache coherence protocol described in the IEEE Futurebus+ standard and the cache consistency protocol developed at Encore Computer Corporation for their Gigamax distributed multiprocessor. Formal verification of embedded systems using symbolic model checking with SMV has been demonstrated in Reference 94: a Petri net-based system model is translated into the SMV input language along with the specification of timing properties.

6.3.2.3.2 Spin [60,95,96]

Spin is a widely distributed software package that supports the formal verification of distributed systems It was developed by the formal methods and verification group at Bell Laboratories.

Spin relies on the high-level specification language PROMELA (Process MetaLanguage), a non-deterministic language based on Dijkstra's guarded command language notation and CSP. PROMELA contains primitives for specifying asynchronous (buffered) message passing via channels with an arbitrary number of message parameters. It also allows for the specification of synchronous message passing systems (rendezvous) and mixed systems, using both synchronous and asynchronous communications. The language can model dynamically expanding and shrinking systems, as new processes and message channels can be created and deleted on the fly. Message channel identifiers can be passed in messages from one process to another.

Correctness properties can be specified as standard system or process invariants (using assertions), or as general Linear Temporal Logic (LTL) requirements, either directly in the syntax of next time free LTL, or indirectly as Büchi Automata (expressed in PROMELA syntax as *never* claims). Spin can be used in three modes: for rapid prototyping with random, guided, or interactive simulation; as an exhaustive verifier,

capable of rigorously proving the validity of user-specified correctness requirements (using partial order reduction to optimize search); and as proof approximation system that can validate very large protocol systems with maximal coverage of the state space.

Spin has been applied to the verification of data transfer and bus protocols, controllers for reactive systems, distributed process scheduling algorithms, fault-tolerant systems, multiprocessor designs, local area network controllers, microkernel design, and many other applications. The tool checks the logical consistency of a specification. It reports on deadlocks, unspecified receptions, and flags incompleteness, race conditions, and unwarranted assumptions about the relative speeds of processes.

6.3.2.3.3 COordination SPecification ANalysis (COSPAN) [97]

The COSPAN is a general purpose, rapid prototyping tool developed at AT&T that provides a theoretically seamless interface between an abstract model and its target implementation, thereby supporting top-down system development and analysis. It includes facilities for documentation, conformance testing, software maintenance, debugging, and statistical analysis, as well as libraries of abstract data types and reusable pretested components.

The COSPAN input language, S/R (selection/resolution), belongs to the omega-regular languages which are expressible as finite-state automata on infinite strings or behavioral sequences. COSPAN is based on homomorphic reduction and refinement of omega-automata, that is, the use of homomorphisms to relate two automata in a process based on successive refinement that guarantees that properties verified at one level of abstraction hold in all successive levels. Reduction of the state space is achieved by exploiting symmetries and modularity inherent in large, coordinating systems. Verification is framed as a language-containment problem: checking consists of determining whether the language of the system automaton is contained in the language of the specification automaton. Omega-automata are particularly well-suited to expressing liveness properties, that is, events that must occur at some finite, but unbounded time.

The COSPAN has been used in the commercial development of both software and hardware systems: high-level models of several communications protocols, for example, the X.25 packet switching link layer protocol, the ITU file transfer and management protocol (FTAM), and AT&T's Datakit universal receiver protocol (URP) level C; verification of a custom VLSI chip to implement a packet layer protocol controller; and analysis and implementation of AT&T's Trunk Operations Provisioning Administration System (TOPAS).

6.3.2.3.4 MEIJE [98]

The MEIJE project at INRIA and the Ecole des Mines de Paris has long investigated concurrency theory and implemented a wide range of tools to specify and verify both synchronous and asynchronous reactive systems. It uses Esterel, a language designed to specify and program synchronous reactive systems, and a graphical notation to describe labeled transition systems.

The tools (graphical editors, model checkers, observer generation) operate on the internal structure of automata combined by synchronized product which are generated either from the Esterel programs or from the graphical representations of these automata. MEIJE supports both explicit representation of the automata supporting model checking and compositional reduction of systems using bisimulation or hiding, as well as implicit representation of the automata favoring verification through observers and forward search for properties to verify. To deal with the large state spaces induced by realistic-sized specifications, the MEIJE tools provide various abstraction techniques, such as behavioral abstraction which replaces a sequence of actions by a single abstract behavior, state compression and encoding in BDD, and on-the-fly model checking. Observers can be either directly written in Esterel or can be generated automatically from temporal logic formulae.

6.3.2.3.5 CADP [99]

The CADP, developed at INRIA and Verimag, is a tool box for designing and verifying concurrent protocols specified in the ISO language LOTOS and to study formal verification techniques. Systems are specified as networks of communicating automata synchronizing through rendevous or labeled transition systems.

The CAESAR compiler translates the behavioral part of LOTOS specifications into a C program that can be simulated and tested or into a labeled transition system to be verified by ALDEBARAN. The latter allows the comparison and reduction of labeled transition systems by equivalences of various strength (such as bisimulation, weak bisimulation, branching bisimulation, observational equivalence, tau bisimulation, or delay bisimulation). Diagnostic abilities provide the user with explanations when the tools failed to establish equivalence between two labeled transition systems. The OPEN environment provides a framework for developing verification algorithms in a modular way, and various tools are included: interactive simulators, deadlock detection, reachability analysis, path searching, on-the-fly model checker to search for safety, liveness, and fairness properties, and a tool for generating test suites.

6.3.2.3.6 Murphi [100]

Murphi is a complete finite-state verification system that has been tested on extensive industrial-scale, examples including cache coherence protocols and memory models for commercially designed multiprocessors.

The Murphi verification system consists of the Murphi compiler, and the Murphi description language for finite-state asynchronous concurrent systems which is loosely based on Chandy and Misra's Unity model and includes user-defined data types, procedures, and parameterized descriptions. A version for synchronous concurrent systems is under development. A Murphi description consists of constant and type declarations, variable declarations, rule definitions, start states, and a collection of invariants. The Murphi compiler takes a Murphi description and generates a C++ program that is compiled into a special-purpose verifier that checks for invariant violations, error statements, assertion violations, deadlock, and liveness. The verifier attempts to enumerate all possible states of the system, while the simulator explores a single path through the state space. Efficient encodings, including symmetry-based techniques, and effective hash-table strategies are used to alleviate state explosion.

6.4 Specifying and Verifying Embedded Systems

The problems of consistency and completeness of requirements viewed as mathematical problems are well known to be algorithmically unsolvable even for notations solely based on the first-order predicate calculus. To overcome these difficulties, we have studied the general form of the requirements used in specific subject domains and developed methods of proving sufficient conditions of consistency and completeness.

Each requirements specification defines a class of systems compatible with the requirements. All systems in this class are defined at least up to bisimulation, that is, systems with the same observable behavior are considered as equal. However, systems often operate in the context of some environment; when requirements describe the properties of the environment into which a system is inserted, the weaker notion of insertion equivalence is used to distinguish different systems. If the class of systems compatible with the requirements is not empty, we consider the requirements to be consistent. If the class contains only one system (up to bisimulation or insertion equivalence, respectively), the requirements are said to be complete.

To represent requirements, we distinguish between a class of logical requirement languages representing behavior in logical form, a class of trace languages, representing behavior in the form of traces, and a class of automata networks languages, representing behavior in terms of states and transitions. The latter are *model-oriented* [101], in that desired properties or behaviors are specified by giving a mathematical model that exhibits those properties. The disadvantage of model-oriented specifications is that they state what should be done or how something should be implemented, rather than the properties that are required. For *property-oriented* [101] languages, each description defines not a single model, but a class of models which satisfy the properties defined by the description. The properties expressed are properties of attributed transition systems representing environments and agents inserted into these environments. Only actions and the values of attributes are observable for inserted agents.

6.4.1 System Descriptions and Initial Requirements

The descriptions of observable entities (attributes) of a system and the external environment, their types, and the parameters they depend on, are captured in first-order predicate logic extended by types and certain predefined predicates such as equality or arithmetic inequality. The signature of the language contains a set of attribute names, where each attribute has a type which is defined using direct or axiomatic definitions. Operations defined on the sets of values of different types are used to construct value expressions (terms) from attribute symbols, variables, and constants. If an expression contains attributes, the value of this expression depends not only on the values of its variables, if any, but also on the current state of the environment. Consequentially, it defines a function on the set of states of an environment.

The language includes the temporal modalities **always** and **sometimes**. Logical statements define properties of the environment, characterize agent actions, or define abstract data types.

Initial requirements describe the initial state as a logical statement. If initial requirements are present, requirements refer only to the states reachable from initial states, which are those states satisfying the initial requirements. To describe initial requirements, we use a temporal modality **initially**.

Axioms about the system are introduced by the form

`let` <*name*> : <*statement*>;

6.4.2 Static Requirements

Static requirements define the change of attributes at any given moment or interval of time depending on the occurrence of events and the previous history of system behavior. Static requirements describe the local behavior of a system, that is, all possible transitions which may be taken from the current state if it satisfies the precondition, after the event forcing these transitions has happened. The general form of static requirements is

`req` <*name*>:[<*prefix*>]([<*precondition*>] -> [`after` <*event description*>] <*postcondition*>);

The precondition is a predicate formula of first-order logic, true before the transition; the postcondition is a predicate formula of first-order logic, true after the transition. Both precondition and postcondition may refer to a set of attributes used in the system. Variables are typed and may occur in precondition, event, and postcondition; they link the values of attributes before and after the transition. Only universal quantifiers are allowed in the quantifier prefix. All attributes occur free in the requirements, but if an attribute depends on parameters, the algebraic expressions substituted for the parameter may contain bound variables.

If the current state of the environment satisfies the precondition, and allows the event, then after performing this action the new state of the system will satisfy the property expressed by the postcondition. This notation corresponds to Hoare-style triples.

Predicates are defined on sets of values, and predicate expressions can be constructed using predicate symbols and value expressions to express properties of states of environments (if they include attributes). The quantifiers **forall** and **exists** can be used, as well as the usual propositional connectives.

The precondition is a logical statement without explicit temporal modalities and describes observable properties of a system state. It may include predefined temporal functionals that depend on the past behavior of a system or its attributes, for example, the duration functional **dur**: if P is a statement, then **dur** P denotes the time passed from the last moment when P became true; its value is a number (real for continuous time and an integer for discrete time).

The event denotes the cause of a transition or a change of attribute values. The simplest case of an event is an action. More complex examples are sequences of actions or finite behaviors (i.e., the event is an algebraic expression generated by actions, prefixing, nondeterministic choice, and sequential and parallel compositions). To describe histories we use product $P * Q$ and iteration $\mathrm{It}(P)$ over logical statements.

The postcondition is a logical statement denoting the property of attribute values after the transition defined by the static requirement has been completed. It cannot include temporal modalities as well as functionals depending on the past.

As an example, consider a system which counts the number of people entering a room. The requirement for an action `enter` could be written as:

$$\texttt{req Enter: Forall(n : int)((num = n)- > after(enter) (num = n + 1));}$$

where `num` is a system attribute representing the number of people in the room. The variable `n` links the value of the attribute `num` before and after the transition caused by the `enter` action.

6.4.3 Dynamic Requirements

Dynamic requirements are arbitrary logical statements including temporal modalities and functionals. They should be the consequences of static requirements and, therefore, dynamic requirements are formulated as calls to the prover to establish a logic statement:

$$\texttt{prove<}\textit{statement}\texttt{>;}$$

We use the temporal modalities **always** and **sometimes**, as well as the temporal functional **dur** and Kleene operations (product and iteration) over logical statements. Temporal statements refer to properties of attributed transition systems with initial states. A logical statement without modalities describes a property of a state of a system while temporal statements describe properties of all states reachable from admissible initial states: if P is a predicate formula, the formula **always** P means that any state reachable from an admissible initial state possesses property P. The formula **sometimes** P means that there exists a reachable state that possesses property P. The notion of reachability can be expressed in set-theoretical terms. Temporal modalities can be translated to first-order logic by introducing quantifiers on states and consequentially, it is possible to use first-order provers for proving properties of environments.

For synchronous systems (systems with explicit time assignments at each state) we introduce the functional (E **time** t) for arbitrary expression E (terms or statements) and integer expression t which denotes the value of Eat time t. Then **always** and **sometimes** are defined as:

$$\textbf{always}\ P \Leftrightarrow (\forall t \geq 0)(P\ \textbf{time}\ t)$$

$$\textbf{sometimes}\ P \Leftrightarrow (\exists t \geq 0)(P\ \textbf{time}\ t)$$

The functional **dur** is defined in the following way [102]:

$$(\textbf{dur}\ P = s)\ \textbf{time}\ t \Leftrightarrow (t \geq s) \wedge \sim (P)\ \textbf{time}\ (t - s) \wedge (\forall s')((t \geq s' > t - s) \rightarrow P\ \textbf{time}\ s')$$

Statements, which do not explicitly mention state or time are considered as referring to an arbitrary current state or an arbitrary current moment of time. Statements with Kleene operations refer to discrete time and are reduced to logical statements as follows:

$$(P_1^* P_2^* \ldots^* P_n)\ \textbf{time}\ t \Leftrightarrow (P_1\ \textbf{time}\ (t - n + 1)) \wedge (P_2\ \textbf{time}\ (t - n + 2)) \wedge \cdots \wedge (P_n\ \textbf{time}\ t)$$

$$\mathrm{It}(P)\ \textbf{time}\ t \Leftrightarrow (\exists s \leq t)(\forall s')((t - s \leq s' \leq t) \rightarrow P\ \textbf{time}\ s')$$

6.4.4 Example: Railroad Crossing Problem

The railroad crossing problem is a well-known benchmark to assess the expressiveness of development techniques for interactive systems. We illustrate the description of a synchronous system (in discrete time)

relying on duration functionals. The problem statement is to develop a control device for a railroad crossing so that safety and liveness conditions are satisfied. This system has three components, as shown in Figure 6.4).

The *n*-track railroad has the following observable attributes: `InCr` is a Boolean variable equal to 1 if a train is at the crossing; `Cmg(`*i*`)` is a Boolean variable equal to 1 if a train is coming on track number *i*. At the moment this attribute becomes equal to 1, the time left until the train will reach the crossing is not less than `d_min` and it remains 1 until the train reaches the crossing. `Cmg(`*i*`)` is an input signal to the controller which has a single output signal `DirOp`. When `DirOp` equals to 1, the gate starts opening, and when it becomes 0, the gate starts closing. The attribute `gate` shows the position of the gate. It is equal to `open` when the gate is completely opened and `closed` if it is completely closed. The time taken for the gate to open is `d_open`, the time taken to close is `d_close`. The requirements text below omits the straightforward static requirements. The dynamic properties of the system are safety and liveness. Safety means that when the train is at the crossing, the gate is closed. Liveness means that the gate will open when the train is at a safe distance (Code 6.1).

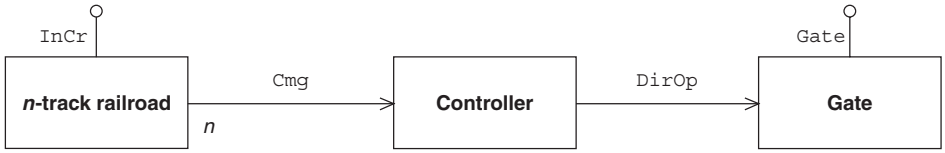

FIGURE 6.4 Railroad crossing problem.

Code 1

```
parameters(
        d_min,
        d_close,
        d_open,
        WT );
    attributes(n:int)(
       InCr:bool,
       Cmg(n):bool,
       DirOp:bool,
       gate );
    let C1:(d_min>d_close);
    let C2:(d_close>0);
    let Duration Theorem: Forall(x,d)(
       always(dur Cmg(x) > d -> ~(DirOp))->
       always(dur ~(DirOp) > dur Cmg(x)+(-1)*(d+1)) );
/* ------------- Environment spec ------------------------ */
    let CrCm:   always(InCr->Exist x (dur Cmg(x) > d_min));
    let OpnOpnd:always( dur   DirOp >d_open ->(gate=opened));
    let ClsClsd:always( dur ~(DirOp)>d_close->(gate=closed));
/* ------------ Controller spec ------------------------ */
    let Contr1: always(Exist  x (dur Cmg(x) > WT ) -> ~(DirOp));
    let Contr2: always(Forall x (WT >= dur Cmg(x)) -> DirOp );
/* ------------- Safety and Liveness --------------------- */
    let(WT=d_min+(-1)*d_close);
    prove Safety:   always(InCr->(gate=closed));
    prove Liveness: always(
       Forall x ((WT > dur Cmg x) -> (gate=opened)));
```

Note the assumption of the Duration Theorem in the requirements to shorten the proofs of safety and liveness.

6.4.5 Requirement Specifications

The example in Section 6.4.4 is rather simple in the number of requirements. Requirement specifications used in practice to describe embedded systems are typically much more complex. Requirement specifications may consist of hundreds or thousands of static requirements, and a large domain descriptions through attributes and parameters. Each requirement is usually simple but taken together the resultant behavior may be complex and contain inconsistencies or be incomplete.

We use attributed transition systems to describe the requirements for embedded systems. The formal specification of requirements consists of the environment description, the description of common system properties in the form of axioms, the insertion function defined by static requirements, and intended properties of the system as a whole defined as dynamic requirements.

A typed list of system parameters and a typed list of system attributes are used to describe the structure of the environment. The *parameters* of the system are variables, which have influence on the behavior of the environment and can change their values from one configuration of the system to another but they *never* change their value during the execution of the system. Examples of system parameters are the set of tasks for an embedded operating system, the bus threshold for a device controller, etc. System *attributes* are variables that differ between the observable states of the environment. Attributes may change their values during runtime. Examples of attributes are the queue of tasks, which are ready to be executed by the operating system, or the current data packet for a device controller.

As an example, we consider (in simplified form) several fragments of the formalized requirements for an embedded operating system for automotive electronics, OSEK [103]. A typed list of system parameters and a typed list of system attributes describes the structure of the environment (Code 6.2).

Code 2

```
parameters (
  tasks: Set of name,
  resources: Set of name
);
attributes (
  suspended: Set of name,
  ready: Set of name,
  running: name
);
```

Parameters of the system are variables, which have influence on the behavior of the environment and can change their values from one configuration of the system to another, but never change their value during the execution of the system.

The operating system (environment) and executing tasks (agents) interact via service calls. The list of actions contains the names of the services defined provided by the system, including service parameters, if any (Code 6.3).

Common system properties are defined as a propositions in first-order logic extended with temporal modalities. For example, consider the following requirements: “the length of the queue of suspended tasks can never be greater than the set of defined tasks.” We will formalize this requirement as follows (Code 6.4).

To define the transitions of the system when processing the request for a service we use Hoare-style triples notation, as defined above (Code 6.5).

Code 3

```
a: name) (
  Activate a,
  Terminate,
  Schedule );
```

Code 4

```
Let SuspendedLengthReq:
  Always ((length(suspended)<|tasks|) |/ (length(suspended) = |tasks|));
```

Code 5

```
req Activate1: Forall (a:name, s: Set of name, r: Set of name) (
  ( (suspended = s) & (ready = r) & (a in s) )
  -> after (Activate a)
  ( (suspended = (s setminus a)) & (ready = (r union a)) ));
```

The insertion function expressed by this rule is sequential, in that only one running task can be performed at a time, all others are in a state suspended or ready. A task becomes running as a result of performing a schedule action. It is selected from a queue of ready tasks ordered by priorities. Agents can change the behavior of the environment by service requests. The interaction between the environment and the agents is defined by an insertion function, which computes the new behavior of the environment with inserted agents.

The part of the description of requirements specific to sequential environments is the definition of the interaction of agents and environments, where this interaction is described by the insertion function. The most straightforward way to define this function is through interactive requirements: an action is allowed to be processed if and only if the current state of the environment matches one of the preconditions for service requests. This is denoted as `E-(act)->E'` intuitively meaning that the environment E allows the action `act` and if it will be processed `then` the environment will be equal to E'.

The agent (composition of all agents) interacting with the environment requests the service `act` if and only if it transits from its current state u into state u' with action `act.u-(act)->u'`.

The composition of environment and the agent is noted as `env(E,u)`. To define the transition of `env(E,u)` we use *interactive rules* (Code 6):

Code 6

```
req ActivateInteract1:
  Forall (E: env, E':env, u:agent, u':agent, a:name)(
    ( (E-(Activate a)->E') & (u-(Activate a)->u') )
    ->
    ( env(E,u) -(Activate a)-> env(E', (Schedule;u')))));
req ActivateInteract2:
  Forall (E: env, E':env, u:agent, u':agent, a:name)(
    (~(E -(Activate a)-> E') & (u-(Activate a)->u') )
    ->
    ( (E'.error = 1) & env(E,u) -(Activate a)-> env(E', bot)) ));
```

Intuitively, this rule means that when the environment allows the action `Activate a` and the agent requests the action `Activate a`, then the composition of agent and environment `env(E,u)` transitions to the state `env(E,(Schedule;u'))` with the action `Activate a'`, then at the next step the environment will be equal to E' and the current agent will be `(Schedule;u')`.

6.4.5.1 Requirements for Sequential Environments

This class of models includes such products as single processor operating systems and single client devices. The definitive characteristic of such systems is that at any moment of time only one service request can be processed by the environment. *Agents* request services from the environment; they are defined by their behavior. The only way of interaction between the environment and the agents is to interact through service requests. It determines the level of abstraction that we use in the formal definition of the behavior of agents.

The insertion functions used for the description of sequential systems is broader than the insertion functions discussed earlier. An inserted agent can start its activity before agents inserted earlier terminate. The active agent can be selected by the environment using various criteria such as priority or other static or dynamic characteristics. To compare agent behaviors, in some cases a look-ahead insertion may be used.

Usually, sequential environments are deterministic systems and static requirements should be consistent to define deterministic transitions. Consistency requirements reduce to the following condition: the preconditions for each pair of static requirements referring to the same action must be nonintersecting. In other words, for arbitrary values of attributes there must be at least one of two requirements which has a false precondition. Completeness can also be checked for the set of all static requirements that refer to the same action. Every such set of requirements must satisfy the condition that for arbitrary values of attributes there must be at least one among the requirements that is applicable with a true precondition.

6.4.5.2 Requirements for Parallel Environments

A parallel environment is an environment with inserted agents that work in parallel and independently. Agents are considered as transition systems. An agent can transition from one state to another by performing an action at any time when the environment accepts this action. Once the agent has completed the action, it transitions into a new state and, in addition, causes a transition of the environment.

As an example, consider the modeling of device interaction protocols. Devices are independent and connected through the environment. They interact by sending and receiving of data packets. The protocol is considered as an algorithm used by devices to interact with other components.

Such a device is an agent in the parallel environment. It is represented as a transition system that can cause transition between states by one of the two actions: sending or receiving a packet that is a parameter of these actions. We formalize such requirements by using the notation of Hoare-style triples.

Asynchronous parallel environments are highly nondeterministic. Such specifications are easily expressed in sequence diagrams or message sequence charts, as shown in Figure 6.5. The preconditions and postconditions are conditions and states on the message sequence diagram, while the actions represent the message arrows and tasks shown on the diagrams.

6.4.5.3 Requirements for Synchronous Agents

As an example of a synchronous system, consider a processor with a bus and its time-dependent behavior. The processor interacts with the bus through signals which appear at every bus cycle (discrete time step). Interaction protocols in the processor-bus system are defined by signal behavior. Every signal can have a value of either 0 or 1. After some event, every signal switches to one of these values. For every signal of the system there is a history describing the conditions of signal switching. Such conditions are called assertion condition (when the signal switches to 1) and deassertion condition (when the signal switches to 0).

Formally, the history of signals is a sequence of conjunctions of signals. The situation when signal S_1 is equal to 1, signal S_2 is equal to 0 in moment t_n and signal S_1 is equal to 0, signal S_2 is equal to 1 in moment t_{n+1} can be described in the following way:

$$(\mathtt{S1} \,\&\, \sim \mathtt{S2}) * (\sim \mathtt{S1} \,\&\, \mathtt{S2})$$

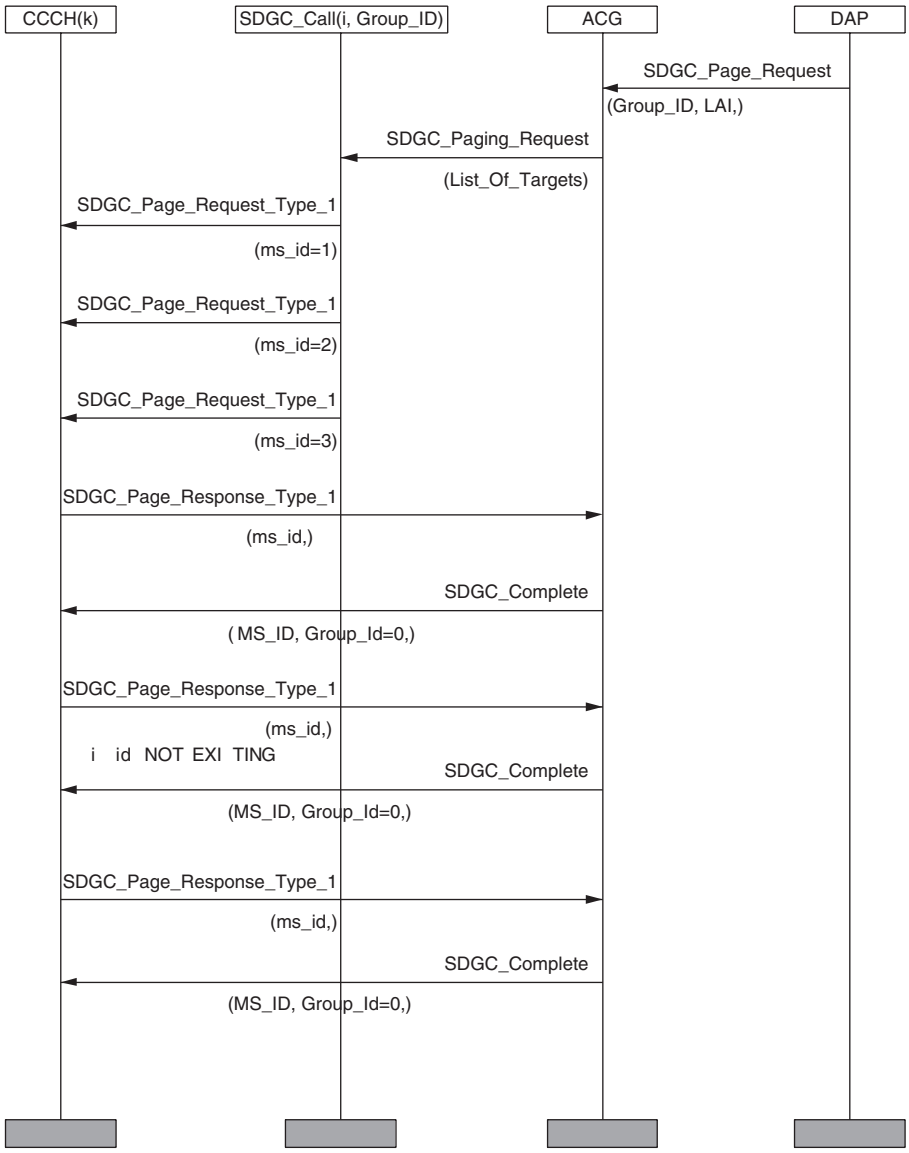

FIGURE 6.5 Sample MSC diagram.

Let signal S_3 have as assertion condition that it will be equal to 1 after the above history of signals S_1 and S_2. This fact can be described as

$$(\mathtt{S1} \,\&\, \sim \mathtt{S2}) * (\sim \mathtt{S1} \,\&\, \mathtt{S2}) - > \mathtt{after(1)}\ \mathtt{S3}$$

using triples notation. This condition can be reflected on a wave (or timing) diagram (Figure 6.6).

The consistency condition is fulfilled if signal q will not be changed into 1 and 0 in the same cycle. In other words, if there are two requirements $P \rightarrow q$ and $P' \rightarrow \sim q$, then preconditions P and P' cannot be true simultaneously.

Static requirements for synchronous systems can use Kleene expressions over conditions and duration functions with numeric inequalities in preconditions. These requirements are converted into standard form with logic statements relating to adjacent time intervals.

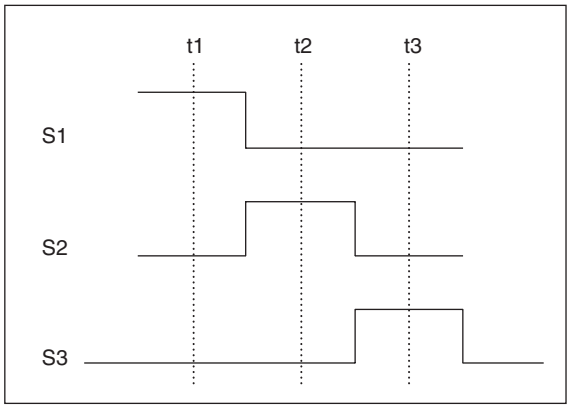

FIGURE 6.6 Sample wave diagram.

6.4.6 Reasoning about Embedded Systems

The theory of agents and environments has been implemented in the system 3CR [104]. The kernel of our system [105] consists of a simulator for a generic Action Language (AL) [10,11] for the description of system behaviors, of services for automatic exploration of the behavior tree of a system, and of a theorem prover for first-order predicate logic, enriched with a theory of linear equations and inequalities. It provides the following technologies supporting the development, verification, and validation of requirements for embedded systems:

- Prove the internal consistency and completeness of static requirements of a system.
- Prove dynamic properties of the system defined by static requirements including safety, liveness, and integrity conditions.
- Translate systems described in standard engineering languages (e.g., MSC, SDL, or wave diagrams) into the first-order format described earlier and simulate these models in user-defined environments.
- Generate test suites for a system defined by verified requirements specifications and validate the implementations of the system against these test cases.

These facilities can be used in automated as well as in interactive mode. To determine consistency and completeness of requirements for interactive systems we rely on the theory of interaction of agents and environments as the underlying formal machinery.

6.4.6.1 Algebraic Programming

The mathematical models described in Section 6.2 can be made more concrete by imposing structure on the state space of transition systems. An universal approach is to consider an algebraic structure of the set of states of a system. Then states are represented by algebraic expressions and transitions can conveniently be defined by (conditional) rewriting rules. A combination of conditional rewriting rules with congruence on the set of algebraic expressions can be defined in terms of rewriting logic [32].

Most modern rewriting techniques are considered primarily in the context of equational theories but could also be applied to first-order or higher-order clausal or nonclausal theorem proving. The main disadvantage of computations with such systems is their relatively weak performance. For instance, rewriting modulo associativity and commutativity (AC-matching) is NP-complete. Consequentially, these systems are usually not powerful enough when "real-life" problems are considered.

Our environment [105] supports reasoning in noncanonical rewriting systems. It is possible to combine arbitrary systems of rewriting rules with different rewrite strategies. The equivalence relation (basic congruence) on a set of algebraic expressions is introduced by means of interpreters for operations

which define a canonical form. The primary strategy of rewriting is a one-step syntactic rewriting with postcanonization by means of reducing the rewritten node to this canonical form. All other strategies are combinations of the primary strategy with different traversals of the tree representing a term structure. Rewrite strategies can be chosen from the library of strategies or written as procedures or functions.

The generic AL [11,106] is used for the syntactical representation of agents as programs and is based on the behavior algebra defined in Section 6.2. The main syntactic constructs of AL are prefixing, nondeterministic choice, sequential composition, and parallel composition. Actions and procedure calls are primitive statements. It provides the standard termination constants (successful termination, divergence, deadlock). The semantics of this language is parameterized by an *intensional semantics* defined through an unfolding function for procedure calls and an *interaction semantics* defined by the insertion function of an environment into which the program will be inserted. The intensional semantics and the interaction semantics are defined as systems of rewriting rules.

The *intensional semantics* of an AL program is an agent which is obtained by unfolding procedure calls in the program and defining transitions on a set of program states. It is defined independently of the environment by means of rewriting rules for the unfolding function (unfolding rules) up to bisimulation. The left-hand side of an unfolding rule is an expression representing a procedure call. The right-hand side of an unfolding rule is an AL program which may be unfolded further generating more and more exact approximations of the behavior under recursive computation.

The only built-in compositions of AL are prefixing and nondeterministic choice. The unfolding of parallel and sequential compositions are flexible and can be adjusted by the user. Alternatives for parallel composition are defined by the choice of the combination operator. For example, when the combination of arbitrary actions is the impossible action, parallel composition is reduced to interleaving. On the other hand, exclusion of interleaving from the unfolding rules defines parallel composition as synchronization at each step (similar to hand shaking in Milner's π-calculus).

The *interaction semantics* of AL programs is defined through the insertion function. Programs are considered up to insertion equivalence. Rewriting rules which define the insertion function (insertion rules) have the following structure: the left-hand side of an insertion rule is the state or behavior of the environment with a sequence of agents inserted into this environment (represented as AL programs). The right-hand side is a program in AL augmented by "calls" to the insertion function denoted as $\mathtt{env}(E, u)$, where E is an environment state expression and u is an AL program. To compute the interaction semantics of AL program one uses both the unfolding rules for procedure calls and the insertion rules to unfold calls to the insertion function.

In this approach, the environment is considered as a semantic notion and is not explicitly included in the agent. Instead, the meaning of an agent is defined as a transformation of an environment which corresponds to inserting the agent into its environment. When the agent is inserted into the environment, the environment changes and this change is considered to be a property of the agent described.

6.4.6.2 Simulating of Transition Systems

The AL has been implemented by means of a simulator [10,106,107], an interactive program which generates all histories of an environment with inserted agents and which can explore the behavior of this environment step-by-step, starting from any possible initial state, with branching at nondeterministic points and backtracking to previous states. The simulator permits forward and backward moves along histories; in automatic mode it can search for states satisfying predefined properties (deadlock, successful termination, etc.) or properties defined by the user. The generation of histories may be user guided and thus permits examination of different histories. The user can retrieve information about the current state of a system and change this state by means of inserting new agents using different insertion functions.

Arbitrary data structures can be used for the representation of the states of an environment and the environment actions. The set of states of an environment is closed under the insertion function $e[u]$ which is denoted in the simulator as $\mathtt{env}(e, u)$. The agent u is represented by an AL expression. Arbitrary algebraic data structures can be used for the representation of agent actions and procedure calls.

The core of the simulator is specified as a nondeterministic transition system that functions as an environment for the system model. Actions of the simulating environment are expressed by means of calls for services of the simulator. Local services define one-step transition of the simulated system. Global services permit the user to compute different properties of the behavior of a simulated system. The user can formulate the property of a state by means of a rewriting rule system or some other predicate function and the simulator will search for the existence of a state satisfying the property among the states reachable from the current state. Examples of such properties are deadlock, successful termination, undefined states, and so on.

6.4.6.3 Theorem Proving

The proof system [108] is based on the interactive evidence algorithm [109–111] — a Gentzen-style calculus with unification used for first-order reasoning.

The Interactive Evidence Algorithm is a sequent calculus and relies on the construction of an auxiliary goal as the main inference step which allows easy control of the direction of the search for proofs at each step through the choice of auxiliary goals. This algorithm can be represented as a combination of two calculi: inference in the calculus of auxiliary goals is used as a single-step inference in the calculus of conditional sequents. In a sense, the interactive evidence algorithm generalizes logic programming in that for the latter, auxiliary goals are extracted from Horn disjuncts while in the interactive evidence algorithm they are extracted from arbitrary formulae with quantifiers (which need not be skolemized).

The interactive evidence algorithm is implemented as a nondeterministic algebraic program extracted from the calculus based on the simulator for AL. This program is inserted as an agent into a control environment which searches for a proof, organizes interaction with the user and the knowledge bases, and implements strategies and heuristics to speed up the proof search. The control environment contains the assumptions of a conditional sequent, and so the local information can be combined with other information taken from knowledge base agents and used in search strategies.

The prover is invoked by the function `prove` implemented as a simple recursive procedure with backtracking which takes an initial conditional sequent as argument and searches for a path from the initial statement to axioms, and this path is converted to a proof. The inference search is nondeterministic owing to disjunction rules.

Predicates are considered up to equivalence defined by means of all Boolean equations except distributivity. A function *Can* defined by means of a system of rewriting rules defines the reduction of predicate formulae as well as propositional formulae to a normal form. Predicate formulae are considered up to renaming of bound variables and equations $\neg(\exists x)p = (\forall x)\neg p$, $\neg(\forall x)p = (\exists x)\neg p$. Associativity, commutativity, and idempotence of conjunction and disjunction as well as the laws of contradiction, excluded middle, and the laws for propositional constants are used implicitly in these equations.

6.4.7 Consistency and Completeness

The notion of consistency of requirements in general is equivalent to the existence of an implementation or model of a system that satisfies these requirements. Completeness means that this model is unique up to some predefined equivalence. The traditional way of proving consistency is to develop a model coded in some programming or simulation language and to prove that this code is correct with respect to requirements. However, direct proving of correctness is difficult because it demands computing necessary invariant conditions for the states of a program. Another method is generating the space of all possible states of a system reachable from the initial states and checking whether the dynamic requirements are satisfied in each state. This approach is known as model checking and many systems which support model checking have been developed. Unfortunately, model checking is realistic only if the state space is finite, and all reachable states can be generated in a reasonable amount of time.

Our approach proves consistency and completeness of requirements directly, without developing a model or implementation of the system. We prove that the static requirements define the system completely and that dynamic properties of consistent requirements are all the logical consequences of static

requirements. Based on this assumption, one can define an executable specification using only static requirements and then execute it using a simulator.

We distinguish between the consistency and completeness of static requirements and dynamic consistency. The first is defined in terms of static requirements only and reflects the property of a system to be deterministic to actions by the environment. For example, a query from a client to a server as the action of an inserted agent can be selected nondeterministically, but the response must be defined by static requirements selected in a deterministic manner. When all dynamic requirements are the consequences of static requirements, we say the system is dynamically consistent.

Sufficient conditions for the consistency of static requirements depend on subject domains and implicit assumptions about the change of observable attributes. For example, for the classes of asynchronous systems considered previously, the condition for internal consistency is simply that the conjunction of two preconditions corresponding to different rules with the same action is not satisfiable. Completeness means that the disjunction of all preconditions for all rules corresponding to the same action is generally valid. For synchronous systems, on the other hand, it is the nonsatisfiability of two preconditions corresponding to rules which define conflicting changes to the same (usually binary) attribute. The incompleteness of static requirements usually is not harmful, it merely postpones design decisions to the implementation stage. However, it is harmful if there exists an implementation which meets the static requirements but does not meet the dynamic requirements.

Dynamic consistency of requirements (the invariance of dynamic conditions expressed using the temporal modality "always") can be proven inductively using the structure of static requirements. Consistency checking proceeds by formulating and proving consistency conditions for every pair of static requirements with the same starting event. Every such pair of requirements must satisfy the condition that for arbitrary values of attributes there must be at least one of the two requirements which has a false precondition or the postconditions are equivalent.

Completeness of requirements means that there exists exactly one model for the requirements up to some equivalence. We distinguish two main cases depending on the focus of the requirements specification. If the specification defines the environment, the equivalence of environments needs to be considered. Otherwise, if an agent is defined by the requirements, the equivalence of agents needs to be examined.

Let e and e' be two environment states (of the same or different environments). We say that e and e' are equivalent if for an arbitrary agent u the states $e[u]$ and $e'[u]$ are bisimilar (from the equivalence of two environment states it follows that for insertion equivalent agents u and u', $e[u]$ and $e'[u']$ are also bisimilar). If there are restrictions on possible behaviors of the agents, we consider admissible agents rather than arbitrary agents.

Let E and E' be two environments (each being a set of environment states and an insertion function). These environments are equivalent if each state of one of the environments is equivalent to some state of the other.

If the set of environments defines an agent for a given environment E, logical completeness (with respect to agent definition) means that all agents satisfying these requirements are insertion equivalent with respect to the environment E, that is, if u and u' satisfy the requirements, then for all $e \in E$, $e[u] \sim_E e[u']$.

We check completeness for the set of all static requirements that refer to the same starting event. Every such set of requirements must satisfy the condition that for arbitrary values of attributes there must be at least one among the requirements that is applicable with a true precondition.

6.5 Examples and Results

Figure 6.7 exhibits a design process using the 3CR [104] tool set. The requirements for a system are represented as input text written in the formal requirements language or translated from engineering notations, such as SDL or MSC. Static requirements are sent to the checker which establishes their consistency and completeness. The checker analyzes a requirement statement and generates a logical

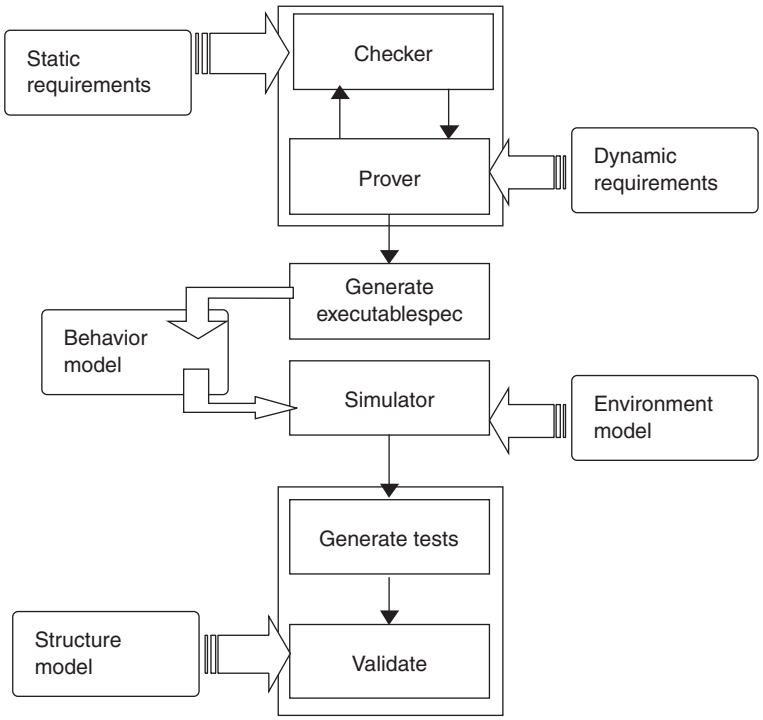

FIGURE 6.7 Design process.

statement expressing the consistency of the given requirement with other requirements already accepted, as well as a statement expressing completeness after all static requirements have been accepted. Then this statement is submitted to the prover in order to search for a proof. The prover may return one of three answers: *proved*, *not proved*, or *unknown*. In the case where consistency could not be proven, one of the following types of inconsistencies is considered.

- *Inconsistent formalization.* This type of inconsistency can be eliminated through improved formalization, if the postconditions are consistent for the states where all preconditions are true. Splitting the requirements can help.
- *Inconsistency resulting from incompleteness.* This is the case when two requirements are consistent, but nonintersection of preconditions cannot be proved because complete knowledge of the subject domain is not available. A discussion with experts or the authors of the requirements is recommended.
- *Inconsistency.* Preconditions are intersected, but postconditions are inconsistent after performing an action. This is a sign of a possible error, which can be corrected only by the change of requirements. If the intersection is not reachable, the inconsistency will not actually arise. In this case, a dynamic property can be formulated and proven.

Dynamic properties are checked after accepting all static requirements. These are logical statements expressing properties of a system in terms of first-order predicate calculus, extended by temporal modalities, as well as higher-order functions and types. If an inductive proof is needed, all static requirements are used for generating lemmas to prove the inductive step.

After checking the consistency and completeness of static requirements, the requirements are used for the automatic generation of an executable specification of a system satisfying the static requirements. At this point, the dynamic requirements have already been proven to be consequences of static requirements, so the system also satisfies the dynamic requirements. The next step of system design would be the use of

the obtained information in the next stages of development. For example, executable specifications can be used for generating complete test cases for system test.

6.5.1 Example: Embedded Operating System

In this section, we shall describe a general model which could be used for developing formal requirements for embedded operating systems such as OSEK [103].

The requirements for the OSEK operating system can serve as an example of the application of the general methodology of checking consistency. These requirements comprise two documents: OSEK Concept and OSEK API. The first document contains an informal description of conformance classes (BCC1, BCC2, ECC1, ECC2, ECC3) and requirements of the main services of the system. The second document refines the requirements in terms of C function headers and types of service calls.

Two kinds of requirements can be distinguished in these documents. *Static* requirements define permanent properties of the operating system, which must be true for arbitrary states and any single-step transition. These requirements refer to the structure of operating system states and their changes in response to the performance of services. *Dynamic* requirements state global system properties such as the absence of deadlocks or priority inversions.

Using the theory of interaction of agents and environments as the formalism for the description of OSEK, an environment consists of a processor (or processor network), an operating system, and the external world which interacts with the environment via some kind of communication network; agents are tasks interacting with the operating system and communication network via services. We use nondeterministic agents over a set of actions representing operating system services as models of tasks. The states of the environment are characterized by a set of observable attributes with actions corresponding to the actions of task agents.

Each attribute defines a partial function from the set E of environment states to the set of values D. E is considered as an algebra with the set of (internal or external) operations defined on it. The domain D should be defined as abstract as possible, for example, by means of set theoretic constructions (functions, relations, powersets) over abstract data types represented as initial algebras, in order to be independent as much as possible of the details of implementation when formulating the requirements specifications.

In monoprocessor systems only one agent is in the active state, that is capturing a processor resource. If e is a state of the environment with no active agents then in the representation $e[u]$ of the environment the state u is a state of an active agent. All other agents are in nonactive states (suspended and ready states for OSEK) and are included into the state e as parts of the values of attributes.

The properties of an environment can be divided into static and dynamic properties. Static properties define one-step transitions of a system; dynamic properties define the properties of the total system. The general form of a rule for transitions is:

$$\frac{e \xrightarrow{c} e', u \xrightarrow{a} u'}{e[u] \xrightarrow{d} e''[u'']}$$

In this rule d, e'', and u'' depend on parameters appearing in the assumptions. Usually, if $a = c = d$ (synchronization), $e'' = e'$, and $u'' = u'$, albeit there can be special cases such as hiding, scheduling points, or interrupt routines.

To define the properties of a transition $e \xrightarrow{c} e'$ for the environment we first define the transition rules for the attributes associating a transition system to each attribute. The states of a system associated to the attribute p is a pair $p : v$ where v is a value of a type associated with the attribute p. All transition systems are defined jointly, that is the transitions of one attribute can depend on the current values or transitions of other ones. After defining the transitions for attributes the transitions for environment states must be

defined in such a way that the following general rule should be true:

Let $p_1, \ldots, p_n$ be attributes of a state e of the environment and $e \cdot p_1 = v_1, \ldots, e \cdot p_n = v_n$. Let $e \xrightarrow{c} e'$, e' is not a terminal state (Δ, $\perp$, or 0), and $p_i : v_i \xrightarrow{c} p_i : v_i'$ for all $i \in I \subseteq [1 : n]$ where I is the set of all indices for which such transitions defined. Then $e' \cdot p_i = v_i'$ for $i \in I$ and $e' \cdot p_i = e \cdot p_i$ for $i \notin I$. From this definition it follows that if $I = \emptyset$ and $e \xrightarrow{c} e'$ then $e' \cdot p_i = e \cdot p_i$ for all $i \in I \subseteq [1 : n]$.

In the case, when two states of the environment are bisimilar, this rule is sufficient to define the transitions of the environment. Otherwise we can introduce the hidden part of the environment state and consider transitions of attributes jointly with this hidden component.

For space considerations, in Section 6.5.1.1 we show only the example of a simple scheduler applicable to this class of operating systems.

6.5.1.1 Requirements Specification for a Simple Scheduler

This example of a simplified operating system providing initial loading and scheduling for tasks and interrupt processing is used as a benchmark to demonstrate the approach for formalizing and checking the consistency of requirements. We use the terminology of OSEK [103].

The attributes of the scheduler are:

- **`Active`**, a name
- **`Priority`**, a partial function from names to natural numbers
- **`Ready`**, a list of name/agent pairs
- **`Call`**, a partial function from names to agents

The empty list and the everywhere undefined function are denoted as **`Nil`**. These attributes are defined only for nonterminal and deterministic states. The actions of task agents are calls for services:

- **`new_task`** (a, i), a is a name of an agent, i is an integer
- **`activate`** a, a is a name
- **`terminate`**
- **`schedule`**

In the following requirements we assume that the current state of the environment is $e[u]$ and that $u \xrightarrow{c} u'$ for a given service c. The values of attributes are their values in a state e. We define the transitions $e[u] \xrightarrow{d} e''[u'']$.

The actions of environment include all task actions and, in addition, the following actions which are specific only for the environment and are addressed to an external observer of scheduler activity:

- **`loaded`** a, a is a name
- **`activated`** a, a is a name
- **`activate_error`**
- **`schedule_error`**
- **`terminated`** a, a is a name
- **`schedule`** u, u is an agent
- **`scheduled`** a, a is a name
- **`wait`**
- **`start_interrupt`**
- **`end_interrupt`**

6.5.1.1.1 Requirements for **`new_task`**

This action substitutes the old task with the same name if it was previously defined in the scheduler or adds the task to an environment as a new task otherwise. Transitions for the attributes:

$$\textbf{priority} : f \xrightarrow{\texttt{new\_task}(a:v,i)} \textbf{priority} : f[a := i]$$

We use the following notation for the redefinition of functions: if $f : X \to Y$ and $x \in X$ then $f[x := y]$ is a new function g such that $g(x) = y$ and $g(x') = f(x')$ for $x \neq x'$ (assignment for functions). If $x \notin X$ it is added to the domain of a function and then an assignment is performed.

$$\texttt{call} : f \xrightarrow{\texttt{new\_task}\ (a{:}v,i)} f[a := v]$$

Now the task agent v becomes the initial state of a task named a. **new_task** is defined by the following rule:

$$\frac{e \xrightarrow{\texttt{new\_task}\ (a{:}v,i)} e',\ u \xrightarrow{\texttt{new\_task}\ (a{:}v,i)} u'}{e[u] \xrightarrow{\texttt{loaded}\ a} e'[u']}$$

6.5.1.1.2 Requirements for activate

We use the following notation: if p is an attribute, its value is a function and x is in the domain of this function, then $p(x)$ denotes the current value of this function on x.

$$\frac{\texttt{call}\ a = v}{\texttt{ready} : r \xrightarrow{\texttt{activate}\ a} \texttt{ready} : \texttt{ord}(a : v, r)}$$

The function **ord** is defined on the set of lists of pairs $(a : u)$ where a is a name and u is an agent and this function must satisfy the following system of axioms where all parameters are assumed to be universally quantified:

$$\texttt{ord}(a : \Delta, r) = r$$

$$\texttt{priority}\ b \leq \texttt{priority}\ a \Rightarrow \texttt{ord}(a : u, b : v, r) = (b : v, \texttt{ord}(a : u, r))$$

Hence **ready** is a queue of task agents ordered by priorities and adding a pair $(a : u)$ put this pair as the last one among all pairs of the same priority as a. The rules are:

$$\frac{e \xrightarrow{\texttt{activate}\ a} e',\ u \xrightarrow{\texttt{activate}\ a} u',\ a \in Dom(\texttt{call})}{e[u] \xrightarrow{\texttt{activated}\ a} e'[u']}$$

$$\frac{u \xrightarrow{\texttt{activate}\ a} u',\ a \notin Dom(\texttt{call})}{e[u] \xrightarrow{\texttt{activate\_error}} \perp}\ -$$

An undefined state of the environment only means that a decision about the behavior of the environment in this case is left for the implementation stage. For instance, the definition can be extended so that the environment sends an error message and calls error processing programs or continuous its functioning ignoring the incorrect action.

6.5.1.1.3 Requirements for Terminate

$$\frac{u \xrightarrow{\texttt{terminate}} u'}{e[u] \xrightarrow{\texttt{activated}\ (e.\text{active})} e[\texttt{schedule}\,]}$$

6.5.1.1.4 Requirements for Schedule

Let $P(u, b, v, s) = P_1 \vee P_2$ where

$$P_1 = e.\texttt{active} \neq \texttt{Nil} \wedge \texttt{ord}(e.\texttt{active} : u, e.\texttt{ready}) = (b : v, s)$$

$$P_2 = e.\texttt{active} = \texttt{Nil} \wedge u = \Delta \wedge e.\texttt{ready} = (b : v, s)$$

Let $r = e.\mathbf{ready}$, and $a = e.\mathbf{active}$, then the rules for attributes are:

$$\frac{P(u, b, v, s)}{\mathbf{ready} : r \xrightarrow{\mathbf{schedule}\ u} \mathbf{ready} : s}$$

$$\frac{P(u, b, v, s)}{\mathbf{active} : a \xrightarrow{\mathbf{schedule}\ u} \mathbf{active} : b}$$

Note that, transitions for attributes and therefore for the environment are highly nondeterministic because the parameter u is an arbitrary agent behavior. But this nondeterminism disappears in the rule for scheduling which restricts the possible values for u to no more than one value. The rules are:

$$\frac{P(u', b, v, s), e \xrightarrow{\mathbf{schedule}\ u'} e', u \xrightarrow{\mathbf{schedule}} u'}{e[u] \xrightarrow{\mathbf{scheduled}\ b} e'[v]}$$

$$\frac{u \xrightarrow{\mathbf{schedule}} u', e.\mathbf{active} = \mathtt{Nil} \wedge u' \neq \Delta}{e[u] \xrightarrow{\mathbf{schedule\_error}} \bot}$$

$$\frac{u \xrightarrow{\mathbf{schedule}} \Delta, e.\mathbf{ready} = \mathtt{Nil}}{e[u] \xrightarrow{\mathbf{wait}} e'[\Delta]}$$

Therefore, if a task has no name (it can happen if a task is initially inserted into an environment) it can use scheduling only as the last action, otherwise it is an error. And if there is nothing to schedule, the scheduling action is ignored.

6.5.1.1.5 Interrupts

The simplest way to introduce interrupts to our model is to hide the occurrence of interrupts and the choice of the start of interrupt processing. Only actions which show the start and the end of interrupt processing are observable. The rules are:

$$\frac{e \xrightarrow{\mathbf{start\_interrupt}} e'[v]}{e \xrightarrow{\mathbf{start\_interrupt}} e'[v; \mathbf{end\_interrupt}\ ; u]}$$

We have no transitions for attributes labeled by the interrupt action so in this transition e and e' have the same values for all attributes. The program v is an interrupt processing routine.

$$\frac{u \xrightarrow{\mathbf{end\_interrupt}} u'}{e[u] \xrightarrow{\mathbf{end\_interrupt}} e[u']}$$

Nesting of interrupts can be of arbitrary depth. The action **end_interrupt** is an environment action but it is used by the inserted agent after interrupt started to show the end of interrupt processing. Therefore, the set of actions for inserted agent is extended, but still it is not an action of an agent before its insertion into the environment.

6.5.1.1.6 Termination

When all tasks are successfully terminated, the scheduler reaches the waiting state:

$$\texttt{active} : a \xrightarrow{\texttt{wait}} \texttt{active} : \texttt{Nil}$$

$$\frac{\texttt{ready} : \texttt{Nil}, e \xrightarrow{\texttt{wait}} e'}{e[\Delta] \xrightarrow{\texttt{wait}} e'[\Delta]}$$

6.5.1.1.7 Dynamic Requirements

A state e of an environment is called initial if $e.\texttt{ready} = e.\texttt{active} = \texttt{Nil}$, and the domains of functions $e.\texttt{priority}$ and $e.\texttt{call}$ are empty. Let E_0 be the set of all states reachable from the initial states. Define E_{n+1}, $n = 0, 1, \ldots$ as a set of all states reachable from the states $e[u]$, where $e \in E_n$ and u is an arbitrary task agent. The set E of admissible states is defined as a union $E = E_0 \cup E_1 \cup \ldots$. Multiple insertion rules show that the insertion function is sequential. Dynamic requirements for environment states are as follows:

- E does not contain the deadlock state 0.
- There are no undefined states in E except for those which result from error actions.
- Tasks are scheduled in FIFO discipline for the tasks of the same priority, tasks of a higher priority are scheduled first and interrupt actions are nested as brackets.

6.5.1.1.8 Consistency

The only nonconstructive transition in the requirements specification of the simple scheduler is the insertion of an arbitrary agent as an interrupt processing routine. If we restrict the corresponding transitions to the selection from some finite set (even nondeterministically) the requirements will be executable.

To prove dynamic properties, first some invariant properties for E (**always** statements) must be proved. Then after their formalization, dynamic properties are inferred from these invariants:

- $Dom(e.\texttt{priority}) = Dom(e.\texttt{call})$
- $(a : u) \in e.\texttt{ready} \Rightarrow a \in Dom(e.\texttt{priority})$
- $e.\texttt{active} \neq \texttt{Nil} \Rightarrow e.\texttt{active} \in Dom(e.\texttt{priority})$
- $e.\texttt{ready}$ is ordered by priority

In the invariants formulated above, e is assumed to be nonterminal.

6.5.1.2 Input Text to the Consistency Checker

The consistency checker accepts static requirements represented in the form of Hoare-style triples and dynamic requirements in the form of logical formulae. Requirements include the description of typed attributes and actions. The following input text is obtained from the description of simple scheduler considered above. It is statically consistent and can be used for proving dynamic properties of the scheduler. Each requirement describes the change of a state of environment with the inserted agent represented as the value of the attribute `active_task`. The value of this attribute is the behavior of a previously inserted agent which is currently active. The predicate `active_task->`$a \cdot u$ is used to represent the transition `active_task` $\xrightarrow{a} u$. The action axiom is needed to prove consistency for action `wait` (Code 6.7).

Code 7

```
attributes(
    active: name,
    priority: name -> Nat,
    ready: list of (name:agent),
    call: name -> agent,
```

```
    active_task: agent
);
actions(a:name,u:agent,i:int)(
    new_task(a:u,i),
    activate a,
    terminate,
    schedule,
    loaded a,
    activated a,
    activate_error,
    schedule_error,
    terminated a,
    schedule u,
    scheduled a,
    wait,
    start_interrupt,
    end_interrupt
);

Let action axiom: Forall x(~(x.Delta = Delta));
Let ord Delta: Forall(a,r)(ord(a:Delta,r) = r);
Let ord: Forall(a,b,u,v,r)(
  (priority b <= priority a) & ~(a = Delta)
    -> (ord(a:u,b:v,r) = (b:v,ord(a:u,r))));
/* ------------ new_task ------------------------------ */
  req new_task: Forall(a:name, (u,v):agent, i:int)(
    (active_task --> new_task(a:v,i).u)
    -> after(loaded a)
    ((active_task = u) & (priority a = i) & (call a = v)));
/* ------------ activate ------------------------------ */
  req activate success: Forall(a:name,(u,v):agent, r:list of(name:agent))(
    ((active_task --> activate a.u) & (ready = r) &(call a = v)
      & ~(v = Nil))
    -> after(activated a)
    (active_task = u & ready = ord(a:v,r)));
  req activate error: Forall(a:name,u:agent)(
    ((active_task --> activate a.u) & (call a = Nil))
    -> after activate_error
    bot);
/* ------------ terminate ----------------------------- */
  req terminate: Forall(a:name, u:agent)(
    ((active_task --> terminate.u) & (active = a))
    -> after(terminated a)
    (active_task = schedule));
/* ------------ schedule ------------------------------ */
  req schedule success active:
    Forall((u,v):agent, a:name,s:list of(name:agent))(
    ((active_task --> schedule.u) & ~(active = Nil) &
      (ord(active:u,ready) = (a:v,s)))
    -> after(scheduled a)
    ((active_task =u) & (active = a) & (ready = s)));
  req schedule success not active:
    Forall(v:agent, a:name,s:list of(name:agent))(
    ( (active_task = schedule) & (active = Nil) & (ready = (a:v,s)))
```

```
      -> after(scheduled a)
      ((active_task =v) & (active = a) & (ready = s)));
    req schedule error: Forall(u:agent)(
      ((active_task --> schedule.u) & (active = Nil) & ~(u = Delta))
      -> after schedule_error
      bot);
    req schedule final: Forall(v:agent, b:name,s:list of(name:agent))(
      ((active_task --> schedule.Delta) & (ready = Nil))
      -> after wait
      (active_task = Delta));
  /* ------------ interrupt ------------------------------ */
    req start interrupt: Forall((u,v):agent)(
      ((active_task = u) & (interrup_process = v))
      -> after start_interrupt
      (active_task = (v;end_interrupt;u)));
    req end interrupt: Forall(u:agent)(
      (active_task --> end_interrupt.u)
      -> after end_interrupt
      (active_task = u));
  /* ------------ termination --------------------------- */
    req termination: Forall(u:agent)(
      (active_task = Delta) & (ready = Nil)
      -> after wait
      (active_task = Delta))
  /* ------------ dynamic properties -------------------- */
  prove always Forall(a:name)( a in_set Dom(priority)<=>a in_set Dom(call);
  prove always Forall(a:name,u:agent)(
     (a:u)in_list(ready)-> a in_set Dom(priority));
  prove always ~(active = Nil)-> active in_set Dom(priority);
  prove always is_ord ready
```

6.5.2 Experimental Results in Various Domains

We have developed specializations for the following subject domains: sequential asynchronous environments, parallel asynchronous environments, and sequential synchronous agents. We have conducted a number of projects in each domain to determine the effectiveness of formal requirements verification.

Figure 6.8 exhibits the performance of our provers. We show the measurements in terms of MSC diagrams, a familiar engineering notation often used to describe embedded systems. The chart on the left shows performance in terms of "arrows," that is, communications between instances on an MSC diagram. We can see that the performance is roughly linear to the number of arrows up to roughly 800 arrows per diagram. Note that, a typical diagram has much less arrows, no more than hundred in most cases. The chart on the right shows that performance is linear in the number of MSC diagrams (of typical size). Jointly, these charts indicate that the system is scalable to realistically sized applications.

6.5.2.1 OSEK

OSEK [103] is a representative example of an asynchronous sequential environment. The OSEK standard defines an open, embedded operating system for automotive electronics.

The OSEK formal model has been described as an environment for application tasks of different types, considered as agents inserted into this environment. The actions common for agents and environment are the services of the operating system. The system is multitasking but has only one processor and only one task is running at any given moment and, therefore, the system is considered to be sequential. The

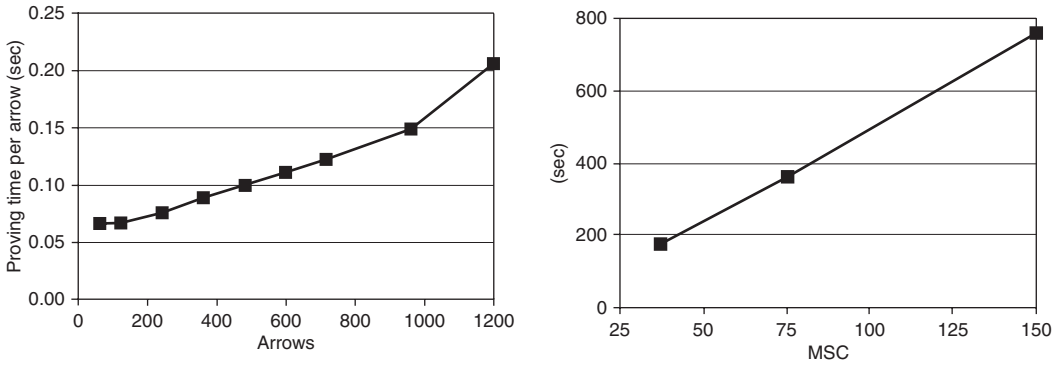

FIGURE 6.8 Performer of prover in terms of MSC diagrams.

system is asynchronous because all actions performed by tasks independently of the operating system are not observable and so the time between two services cannot be taken into account. Static requirements are represented by transition rules with preconditions and postconditions. The reachable states for OSEK can be characterized by integrity conditions.

After developing the formal requirements for OSEK, the proof system was used to prove static consistency and completeness of the requirements. Several interesting dynamic properties of the requirements were also proven. The formalization of OSEK requirements led to the discovery of 12 errors in the nonformal OSEK standard. For example, Section 6.7.5 of the OSEK/VDX specification [103] defines a transition related to the current priority of a task in the case when it has a priority less than the ceiling priority of the resource; however, no transition is defined in the case when the current priority of the task is equal to the ceiling priority.

All these errors were documented and the corrections have been integrated into the OSEK standard. In the formal specification, we have covered 10 services defined by the OSEK standard and have proven the consistency and completeness of this specification. This covers approximately 40% of the complete OSEK standard. Moreover, we have found a number of mistakes in the other parts of the OSEK standard, which prevented formalization of the rest of the standards document.

Consistency and completeness of the covered parts of the standard (49 requirements) were proven, after making corrections for the above mentioned defects. The proof of consistency took approximately 7 min on a Pentium III computer with 256M of RAM running the Red Hat Linux Operating System.

6.5.2.2 RIO

The RapidIO Interconnect Protocol [112] is an example of a parallel asynchronous environment. This is a protocol for a set of processor elements to communicate amongst each other. There are three layers of abstraction developed: logic, transport, and physical layers.

The static requirements for RIO are standard (pre- and postconditions referring to the adjacent moments of time). But while in OSEK an action is uniquely defined by a running task, in RIO it is generated by a nondeterministic choice of one of the processor elements that generates an observable action.

The formal requirements description of RIO for logic (14 requirements) and transport layers (6 requirements) was obtained from the documentation and proved to be consistent and complete (46 sec); 46 requirements for the physical layer have been proven consistent in 8.5 min.

6.5.2.3 V'ger

The formal requirements for the protocol used by the SC-V'ger processor [113] for communicating with other processor elements of a system via the MAXbus bus device were extracted from the documentation of the MAXbus and from discussions with experts. V'ger is a representative example of a synchronous

sequential agent inserted into a parallel environment. V'ger is a deterministic automaton with binary input–output signals and shared data available from the bus. The attributes of the system are its input–output signals and its shared data. Originally, there are no actions and we can consider the clock signal synchronizing the system as the only observable action. Static requirements are written using assertion/deassertion conditions for output signals. Each requirement is a rule for setting the signal to a given value (0 or 1). The precondition is a history of conditions represented in a Klenee-like algebra with time. Several rules can be applied at the same moment. For the static consistency conditions, the preconditions of two rules which set the same attribute to different values can never be true at the same lock interval. There are no static completeness conditions because we define the semantics of the requirements text so that if there are no rules to change the output value, it remains in the same state as in the previous moment of time. We use binary attribute symbols as predicates and as long as there are no other predicate symbols the systems represents a propositional calculus.

To prove statements with Klenee algebra expressions, these must first be reduced to first-order logic, that is, to requirements with preconditions referring to one moment of time (without histories). A converter has been developed for the automatic translation of subject domains relying on Kleene algebra and the interval calculus notation.

The set of reachable states of V'ger is not defined in first-order logic, and the proof of the consistency condition is only a sufficient condition for consistency. A more powerful yet still sufficient condition is the provability of consistency conditions by standard induction from static requirements. There exists a sequence of increasingly powerful conditions which converge to the results obtained by model checking. All 26 V'ger requirements have been proven to be consistent (192 sec).

6.6 Conclusions and Perspectives

In this chapter, we reviewed tools and methods to ensure that the "right" system is developed, by which we mean a system that matches what the customer really wants. Systems that do not match customer requirement result in cost overruns owing to later changes of the system at best, and, in the worst case, may never be deployed. Based on the mathematical model of the theory of agents and interactions we developed a set of tools capable of establishing the consistency and completeness of system requirements. Roughly speaking, if the requirements are consistent, an implementation which meets the requirements is possible; if the requirements are complete, this implementation is defined uniquely by the requirements. We discuss how to represent requirements specifications for formal validation and exhibit experimental results of deploying these tools to establish the correctness of embedded software systems. This chapter also reviews other models of system behavior and other tools for system validation and verification.

Our experience has shown that dramatic quality improvements are possible through formal validation and verification of systems under development. In practice, deployment of these techniques will require increased upstream development effort: thorough analysis of requirements and their capture in specification languages result in a longer design phase. In addition, significant training and experience are needed before significant benefits can be achieved. Nevertheless, the improvements in quality and reduction in effort in later development phases warrant this investment, as application of these methods in pilot projects has demonstrated.

References

[1] D. Harel and A. Pnueli. On the development of reactive systems. In K. Apt, Ed., *Logics and Models of Concurrent Systems*. NATO ASI Series, vol. 13 Springer-Verlag, pp. 477–498.

[2] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, Heidelberg, 1992.

[3] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, Heidelberg, 1995.

[4] F.P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, Reading, MA, 1995.
[5] L. Lamport. Introduction to TLA. SRC Technical note 1994-001, 1994.
[6] R.J. van Glabbeek. Notes on the methodology of CCS and CSP. *Theoretical Computer Science*, 177: 329–349, 1997.
[7] D.M.R. Park. Concurrency and automata on infinite sequences. In *Proceedings of the 5th GI Conference. Lecture Notes in Computer Science*, vol. 104. Springer-Verlag, Heidelberg, 1981.
[8] R. Milner. *Communication and Concurrency*. Prentice Hall, New York, 1989.
[9] J.V. Kapitonova and A.A. Letichevsky. On constructive mathematical descriptions of subject domains. *Cybernetics*, 4: 408–418, 1988.
[10] A.A. Letichevsky and D.R. Gilbert. Towards an implementation theory of nondeterministic concurrent languages. *Second Workshop of the INTAS-93-1702 Project: Efficient Symbolic Computing*, St Petersburg, October 1996.
[11] A.A. Letichevsky and D.R. Gilbert. A general theory of action languages. *Cybernetics and System Analysis*, 1: 12–31, 1998.
[12] R. Milner. The polyadic π-calculus: a tutorial. In F.L. Bauer, W. Brauer, and H. Schwichtenberg, Eds., *Logic and Algebra of Specification*. Springer-Verlag, Heidelberg, 1993, pp. 203–246.
[13] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, New York, 1985.
[14] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60: 109–137, 1984.
[15] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 872–923, 1994.
[16] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on the Foundations of Computer Science*, November 1977, pp. 46–52.
[17] E. Emerson and J. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. *Journal of Computer and System Science*, 30: 1–24, 1985.
[18] M.J. Fisher and R.E. Ladner. Propositional modal logic of programs. In *Proceedings of the 9th ACM Annual Symposium on Theory of Computing*, pp. 286–294.
[19] E. Emerson. Temporal and modal logic. In J. van Leeuwen, Ed., *Handbook of Theoretical Computer Science*. MIT Press, Cambridge, MA, 1991, pp. 997–1072.
[20] R. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35: 677–691.
[21] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 10–20 states and beyond. *Information and Computation*, 98: 142–170.
[22] E. Clarke and E. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *The Workshop on Logic of Programs. Lecture Notes in Computer Science*, vol. 131. Springer-Verlag, Heidelberg, pp. 128–143.
[23] J. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th International Symposium on Programming*, pp. 142–158.
[24] L. Lamport. What good is temporal logic? In R. Mason, Ed., *Information Processing-83: Proceedings of the 9th IFIP World Computer Congress*, Elsevier, 1983, pp. 657–668.
[25] M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15: 73–132, 1993.
[26] W. Thomas. Automata on infinite objects. In J. van Leeuwen, Ed., *Handbook of Theoretical Computer Science*. MIT Press, Cambridge, MA, 1991, pp. 131–191.
[27] A.P. Sistla, M. Vardi, and P. Wolper. The complementation problem for Büchi automata with application to temporal logic. *Theoretical Computer Science*, 49: 217–237.
[28] M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the 1st IEEE Symposium on Logic in Computer Science*, pp. 332–344.
[29] H. Rodgers. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, New York, 1967.

AQ: Please provide volume number for Ref. [15].

AQ: Please provide the year of publication for Refs. [18, 21, 22, 27, 36 50 and 122].

AQ: Please provide the volume number for Ref. [30].

[30] Y. Gurevich. Evolving algebras: an attempt to discover semantics. *Current Trends in Theoretical Computer Science*, 266–292, 1993.
[31] Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, Ed., *Specification and Validation Methods*. University Press, 1995, pp. 9–36.
[32] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96: 73–155, 1992.
[33] P. Lincoln, N. Marti-Oliet, and J. Meseguer. Specification, transformation and programming of concurrent systems in rewriting logic. In G. Bleloch et al., Eds., *Proceedings of the DIMACS Workshop on Specification of Parallel Algorithms American Mathematical Society*, Providence, 1994.
[34] M. Clavel. Reflection in General Logics and Rewriting Logic with Application to the Maude Language. Ph.D. thesis, University of Navarra, 1998.
[35] M. Clavel and J. Meseguer. Axiomatizing reflective logics and languages. In G. Kicrales, Ed., *Reflection'96*. 1996, pp. 263–288.
[36] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Towards Maude 2.0. In F. Futatsugi, Ed., *Proceedings of the 3rd International Workshop on Rewriting Logic and its Applications. Notes in Theoretical Computer Science*, vol. 36, Elsevier, 2000.
[37] J. Meseguer and P. Lincoln. Introduction in Maude. Technical report, SRI International, 1998.
[38] J. Brackett. Software Requirements. Technical report SEI-CM-19-1.2, Software Engineering Institute, 1990.
[39] B. Boehm. Industrial software metrics top 10 list. *IEEE Software*, 4: 84–85, 1987.
[40] B. Boehm. *Software Engineering Economics*. Prentice Hall, New York, 1981.
[41] J.C. Kelly, S.S. Joseph, and H. Jonathan. An analysis of defect densities found during software inspections. *Journal of Systems Software*, 17: 111–117, 1992.
[42] R. Lutz. Analyzing requirements errors in safety-critical embedded sytems. In *IEEE International Symposium Requirements Engineering*, San Diego, 1993, pp. 126–133.
[43] T. DeMarco. *Structured Analysis and System Specification*. Yourdon Press, New York, 1979.
[44] C.V. Ramamoorthy, A. Prakash, W. Tsai, and Y. Usuda. Software engineering: problems and perspectives. *Computer*, 17: 191–209, 1984.
[45] M.E. Fagan. Design and code inspections to reduce errors in program evelopment. *IBM Systems Journal*, 15: 182–211, 1976.
[46] M.E. Fagan. Advances in software inspection. *IEEE Transactions on Software Engineering*, 12: 744–751, 1986.
[47] J. Rushby. Formal Methods and their Role in the Certification of Critical Systems. Technical report CSL-95-1, March 1995.
[48] C.B. Jones. *Systematic Software Development Using VDM*. Prentice Hall, New York, 1990.
[49] J.M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics*. Cambridge University Press, London, 1988.
[50] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, London, 1996.
[51] International Organization for Standardization — Information Processing Systems — Open Systems Interconnection. Lotos — A Formal Description Technique Based on the Temporal Ordering of Observational Behavior. ISO Standard 8807. Geneva, 1988.
[52] R.S. Boyer and J.S. Moore. *A Computational Logic Handbook*. Academic Press, New York, 1988.
[53] M.J.C. Gordon and T.F. Melham, Eds., *Introduction to HOL*. Cambridge University Press, London, 1993.
[54] D. Craigen, S. Kromodimoeljo, I. Meisels, B. Pase, and M. Saaltink. EVES: an overview. In *VDM'91: Formal Software Development Methods. Lecture Notes in Computer Science*, vol. 551. Springer-Verlag, Heidelberg, 1991, pp. 389–405.
[55] M. Saaltink, S. Kromodimoeljo, B. Pase, D. Craigen, and I. Meisels. Data abstraction in EVES. In *Formal Methods Europe'93*, Odense, April 1993.

[56] S. Owre, N. Shankar, and J.M. Rushby. User Guide for the PVS Specification and Verification System. Technical report, SRI International, 1996.
[57] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Cambridge, MA, 2000.
[58] P. Godefroid. VeriSoft: A tool for the automatic analysis of concurrent reactive software. In *Proceedings of the 9th Conference on Computer Aided Verification. Lecture Notes in Computer Science*, vol. 1254. Springer-Verlag, Heidelberg, 1997, pp. 476–479.
[59] J. Burch, E. Clarke, D. Long, K. McMillan, and D. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design*, 13, 1994.
[60] G. Holzmann. *The SPIN Model Checker, Primer and Reference Manual*. Addison-Wesley, Reading, MA, 2004.
[61] S.J. Garland and J.V. Guttag. A Guide to LP, the Larch Prover. Technical report, DEC Systems Research Center Report 82, 1991.
[62] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A tutorial introduction to PVS. In *WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques*. Boca Raton, FL, April 1995.
[63] S. Rajan, N. Shankar, and M. Srivas. An integration of model checking with automated proof checking. In *Proceedings of the 7th International Conference on Computer Aided Verification — CAV '95. Lecture Notes in Computer Science*, vol. 939. Springer-Verlag, Heidelberg, 1995, pp. 84–97.
[64] B. Berard, Ed., *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer-Verlag, Heidelberg, 2001.
[65] International Telecommunications Union. Recommendation Z.120 — Message Sequence Charts. Geneva, 2000.
[66] Object Management Group. Unified Modeling Language Specification, 2.0. 2003.
[67] J. Hooman. Towards formal support for UML-based development of embedded systems. In *Proceedings of the 3rd PROGRESS Workshop on Embedded Systems*, 2002, pp. 71–76.
[68] M. Bozga, J. Fernandez, L. Ghirvth, S. Graf, J.P. Krimm, L. Mounier, and J. Sifakis. IF: an intermediate representation for SDL and its applications. In *Proceedings of the 9th SDL Forum, Montreal*, June 1999.
[69] F. Regensburger and A. Barnard. Formal verification of SDL systems at the Siemens mobile phone department. In *Tools and Algorithms for the Construction and Analysis of Systems — ACAS'98. Lecture Notes in Computer Science*, vol. 1384. Springer-Verlag, Heidelberg, 1998, pp. 439–455.
[70] O. Shumsky and L. J. Henschen. Developing a framework for verification, simulation and testing of SDL specifications. In M. Kaufmann and J.S. Moore, Eds., *Proceedings of the ACL2 Workshop 2000*, Austin, 2000.
[71] P. Baker, P. Bristow, C. Jervis, D. King, and B. Mitchell. Automatic generation of conformance tests from message sequence charts. In *Proceedings of the 3rd SAM (SDL And MSC) Workshop, Telecommunication and Beyond, Aberystwyth. Lecture Notes in Computer Science*, p. 2599, 2003.
[72] B. Mitchell, R. Thomson, and C. Jervis. Phase automaton for requirements scenarios. In *Proceedings of the Feature Interactions in Telecommunications and Software Systems*, vol. VII, 2003, pp. 77–87.
[73] L. Philipson and L. Hogskola. Survey compares formal verification tools. EETIMES, 2001. http://www.eetimes.com/story/OEG20011128S0037
[74] S. Yovine. Kronos: A verification tool for real-time systems. *International Journal of Software Tools for Technology Transfer*, 1: 123–133, 1997.
[75] P. Pettersson and K. Larsen. UPPAAL2k. *Bulletin of the European Association for Theoretical Computer Science*, 70: 40–44, 2000.
[76] D. Bjorner and C.B. Jones, Eds., The Vienna development method: the meta-language. In *Logic Programming. Lecture Notes in Computer Science*, vol. 61. Springer-Verlag, Heidelberg, 1978.
[77] Y. Ledru and P.-Y. Schobbens. Applying VDM to large developments. *ACM SIGSOFT Software Engineering Notes*, 15: 55–58, 1990.

AQ: Please provide the page number for Ref. [59].

[78] A. Puccetti and J.Y. Tixadou. Application of VDM-SL to the development of the SPOT4 programming messages generator. *FM '99: World Congress on Formal Methods, VDM Workshop*, Toulouse, 1999.

[79] J.C. Bicarregui and B. Ritchie. Reasoning about VDM developments using the VDM support tool in Mural. In *VDM 91: Formal Software Development Methods. Lecture Notes in Computer Science*, vol. 551. Springer-Verlag, Heidelberg, 1991, pp. 371–388.

[80] A. Diller. Z: *An Introduction to Formal Methods.* John Wiley & Sons, New York, 1990.

[81] W. Grieskamp, M. Heisel, and H. Dorr. Specifying embedded systems with statecharts and Z: an agenda for cyclic software components. In *Proceedings of the Formal Aspects of Software Engineering — FASE '98. Lecture Notes in Computer Science*, vol. 1382. Springer-Verlag, Heidelberg, 1998.

[82] D. Bert, S. Boulmé, M.-L. Potet, A. Requet, and L. Voisin. Adaptable translator of B specifications to embedded C programs. In *Formal Methods 2003. Lecture Notes in Computer Science*, vol. 2805. Springer-Verlag, Heidelberg, 2003, pp. 94–113.

[83] R. Milne. The Semantic Foundations of the RAISE Specification Language. RAISE report REM/11, STC Technology, 1990.

[84] M. Nielsen, K. Havelund, K. Wagner, and C. George. The RAISE language, methods, and tools. *Formal Aspects of Computing*, 1: 85–114, 1989.

[85] T. Mossakowski, Kolyang, and B. Krieg-Bruckner. Static semantic analysis and theorem proving for CASL. In F. Parisi Presicce, Ed., *Proceedings of the 12th Workshop on Algebraic Development Techniques. Lecture Notes in Computer Science*, vol. 1376. Springer-Verlag, Heidelberg, 1998, pp. 333–348.

[86] P.D. Mosses. COFI: the common framework initiative for algebraic specification and development. In *TAPSOFT'97: Theory and Practice of Software Development. Lecture Notes in Computer Science.* vol. 1214. Springer-Verlag, Heidelberg, 1997, pp. 115–137.

[87] B. Krieg-Brückner, J. Peleska, E. Olderog, and A. Baer. The UniForM workbench, a universal development environment for formal methods. In J. Wing, J. Woodcock, and J. Davies, Eds., *FM'99, Formal Methods. Lecture Notes in Computer Science*, vol. 1709. Springer-Verlag, Heidelberg, 1999, pp. 1186–1205.

[88] C.L. Heitmeyer, J. Kirby, and B. Labaw. Tools for formal specification, verification and validation of requirements. In *Proceedings of the 12th Annual Conference on Computer Assurance, Gaithersburg*, June 1997.

AQ: Please provide the age numbers for Ref. [89].

[89] S. Easterbrook, R. Lutz, R. Covington, Y. Ampo, and D. Hamilton. Experiences using lightweight formal methods for requirements modeling. *IEEE Transactions on Software Engineering*, 24, 1998.

[90] L.C. Paulson. *Isabelle: A Generic Theorem Prover. Lecture Notes in Computer Science*, vol. 828. Springer-Verlag, Heidelberg, 1994, pp. 23–34.

[91] B.J. Krämer and N. Völker. a highly dependable computer architecture for safety-critical control applications. *Real-Time Systems Journal*, 13: 237–251, 1997.

[92] D. Muthiayen. Real-Time Reactive System Development — A Formal Approach Based on UML and PVS. Technical report, Concordia University, 2000.

[93] P.B. Jackson. *The Nuprl Proof Development System, Reference Manual and User Guide.* Cornell University, Ithaca, NY, 1994.

[94] L. Cortes, P. Eles, and Z. Peng. Formal coverification of embedded systems using model checking. In *Proceedings of the 26th EUROMICRO Conference*, Maastricht, September 2000, pp. 106–113.

[95] G. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, New York, 1991.

[96] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Enginering*, 23: 3–20, 1997.

[97] R. Kurshan. *Automata-Theoretic Verification of Coordinating Processes.* Princeton University Press, Princeton, NJ, 1993.

[98] R. de Simone and M. Lara de Souza. Using partial-order methods for the verification of behavioural equivalences. In G. von Bochmann, R. Dssouli, and O. Rafiq, Eds., *Formal Description Techniques VIII*, 1995.

[99] J. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP: a protocol validation and verification toolbox. In *Proceedings of the 8th Conference on Computer-Aided Verification*. New Brunswick, August 1996, pp. 437–440.

[100] D. Dill, A. Drexler, A. Hu, and C. Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*. October 1992, pp. 522–525.

[101] E. Astegiano and G. Reggio. Formalism and method. *Theoretical Computer Science*, 236: 3–34, 2000.

[102] Z. Chaochen, C.A.R. Hoare, and A.P. Ravn. A calculus of durations. *Information Processing Letter*, 40: 269–276, 1991.

[103] OSEK Group. OSEK/VDX. Operating System.Version 2.1. May 2000.

[104] S.N. Baranov, V. Kotlyarov, J. Kapitonova, A. Letichevsky, and V. Volkov. Requirement capturing and 3CR approach. In *Proceedings of the 26th International Computer Software and Applications Conference, Oxford*, 2002, pp. 279–283.

[105] J.V. Kapitonova, A.A. Letichevsky, and S.V. Konozenko. Computations in APS. *Theoretical Computer Science*, 119: 145–171, 1993.

[106] D.R. Gilbert and A.A. Letichevsky. A universal interpreter for nondeterministic concurrent programming languages. In M. Gabbrielli, Ed., *Fifth Compulog Network Area Meeting on Language Design and Semantic Analysis Methods*, September 1996.

[107] T. Valkevych, D.R. Gilbert, and A.A. Letichevsky. A generic workbench for modelling the behaviour of concurrent and probabilistic systems. In *Workshop on Tool Support for System Specification, Development and Verification, TOOLS98*, Malente, June 1998.

[108] A.A. Letichevsky, J.V. Kapitonova, and V.A. Volkov. Deductive tools in algebraic programming system. *Cybernetics and System Analysis*, 1: 12–27, 2000.

[109] A. Degtyarev, A. Lyaletski, and M. Morokhovets. Evidence algorithm and sequent logical inference search. In H. Ganzinger, D. McAllester, and A. Voronkov, Eds., *Logic for Programming and Automated Reasoning (LPAR'99). Lecture Notes in Computer Science*, vol. 1705. Springer-Verlag, 1999, pp. 44–61.

[110] V.M. Glushkov, J.V. Kapitonova, A.A. Letichevsky, K.P. Vershinin, and N.P. Malevanyi. Construction of a practical formal language for mathematical theories. *Cybernetics*, 5: 730–739, 1972.

[111] V.M. Glushkov. On problems of automata theory and artificial intelligence. *Cybernetics*, 5: 3–13, 1970.

[112] Motorola. *RIO Interconnect Globally Shared Memory Logical Specification*. Motorola, 1999.

[113] Motorola. *SC-V'ger Microprocessor Implementation Definition*. Motorola, 1997.

[114] S. Abramsky. A domain equation for bisimulation. *Information and Computation*, 92: 161–218, 1991.

[115] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126: 183–235, 1994.

[116] S.N. Baranov, C. Jervis, V. Kotlyarov, A. Letichevsky, and T. Weigert. Leveraging UML to deliver correct telecom applications. In L. Lavagno, G. Martin, and B. Selic, Eds., *UML for Real: Design of Embedded Real-Time Systems*. Kluwer Academic Publishers, Amsterdam, 2003.

[117] J. Bicarregui, T. Dimitrakos, B. Matthews, T. Maibaum, K. Lano, and B. Ritchie. The VDM+B project: objectives and progress. In *World Congress on Formal Methods in the Development of Computing Systems*. Toulouse, September 1999.

[118] G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modeling Language User Guide*. Addison-Wesley, Reading, MA, 1997.

[119] S. Chandra, P. Godefroid, and C. Palm. Software model checking in practice: an industrial case study. In *Proceedings of the International Conference on Software Engineering*, Orlando, May 2002.

AQ: Refs. [114–151] have not cited in text.

[120] E. Clarke, I. Draghicescu, and R. Kurshan. A Unified Approach for Showing Language Containment and Equivalence between Various Types of Omega-Automata. Technical report, Carnegie-Mellon University, 1989.
[121] F. Van Dewerker and S. Booth. Requirements Consistency — A Basis for Design Quality. Technical report, Ascent Logic, 1998.
[122] E. Felt, G. York, R. Brayton, and A. Vincentelli. Dynamic variable reordering for BDD minimization. In *Proceedings of the EuroDAC*, pp. 130–135.
[123] M. Fitting. A Kripke-Kleene semantics for logic programs. *Journal of Logic Programming*, 2: 295–312, 1985.
[124] I. Graham. *Migrating to Object Technology*. Addison-Wesley, Reading, MA, 1995.
[125] Green Mountain Computing Systems. Green Mountain VHDL Tutorial, 1995.
[126] International Telecommunications Union. Recommendation Z.100 — Specification and Description Language. Geneva, 1999.
[127] B. Jacobs. Objects and classes, coalgebraically. In B. Freitag, C.B. Jones, C. Lengauer, and H.-J. Schek, Eds., *Object-Orientation with Parallelism and Persistence*. Kluwer Academic Publishers, 1996, pp. 83–101.
[128] I. Jacobson. *Object-Oriented Software Engineering, A Use Case Driven Approach*. Addison-Wesley, Reading, MA, 1992.
[129] N.D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, New York, 1993.
[130] J.V. Kapitonova, T.P. Marianovich, and A.A. Mishchenko. Automated design and simulation of computer systems components. *Cybernetics and System Analysis*, 6: 828–840, 1997.
[131] M. Kaufmann and J.S. Moore. ACL2: an industrial strength version of NQTHM. In *Proceedings of the 11th Annual Conference on Computer Assurance (COMPASS96)*, June 1996, pp. 23–34.
[132] S. Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16: 83–94, 1963.
[133] J. van Leeuwen, Ed., *Handbook of Theoretical Computer Science*. MIT Press, Cambridge, MA, 1991.
[134] A.A. Letichevsky, and J.V. Kapitonova. Mathematical information environment. In *Proceedings of the 2nd International THEOREMA Workshop*, Linz, June 1998, pp. 151–157.
[135] A.A. Letichevsky and D.R. Gilbert. Agents and environments. In *Proceedings of the 1st International Scientific and Practical Conference on Programming*, Kiev, 1998.
[136] A.A. Letichevsky and D.R. Gilbert. A model for interaction of agents and environments. In *Selected Papers from the 14th International Workshop on Recent Trends in Algebraic Development Techniques. Lecture Notes in Computer Science*. vol. 1827, 2004, pp. 311–328.
[137] P. Lindsay. On transferring VDM verification techniques to Z. In *Proceedings of Formal Methods Europe — FME'94*, Barcelona, October 1994.
[138] W. McCune. Otter 3.0 Reference Manual and Guide. Technical report, Argonne National Laboratory Report ANL-94, 1994.
[139] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Dordrecht, 1993.
[140] M. Morockovets and A. Luzhnykh. Representing mathematical texts in a formalized natural like language. In *Proceedings of the 2nd International THEOREMA Workshop, Linz*, June 1998, pp. 157–160.
[141] T. Nipkow, L. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic. Lecture Notes in Computer Science*, vol. 2283. Springer-Verlag, Heidelberg, 2002.
[142] S. Owre, J.M. Rushby, and N. Shankar. A prototype verification system. In D. Kapur, Ed., *Proceedings of the 11th International Conference on Automated Deduction (CADE). Lecture Notes in Artificial Intelligence*, vol. 601, Springer-Verlag, Heidelberg, 1992, pp. 748–752.
[143] G. Plotkin. A Structured Approach to Operational Semantics. Technical report, DAIMI FN-19, Aarhus University, 1981.
[144] K.S. Rubin and A. Goldberg. Object behavior analysis. *Communications of the ACM*, 35: 48–62, 1992.

[145] R. Rudell. Dynamic variable reordering for ordered binary decision diagrams. In *Proceedings of the IEEE/ACM ICCAD'93*, 1993, pp. 42–47.

[146] J. Rushby. Mechanized formal methods: where next? In J. Wing and J. Woodcock, Eds., *FM99: The World Congress in Formal Methods. Lecture Notes in Computer Science*, vol. 1708. Springer-Verlag, Heiderberg, 1999, pp. 48–51.

[147] J. Rushby, S. Owre, and N. Shankar. Subtypes for specifications: predicate subtypes in PVS. *IEEE Transactions on Software Engineering*, 24: 709–720, 1998.

[148] M. Saeki, H. Horai, and H. Enomoto. Software development process from natural language specification. In *International Conference on Software Engineering*. Pittsburgh, March 1989, pp. 64–73.

[149] J. Tsai and T. Weigert. *Knowledge-Based Software Development for Real-Time Distributed Systems.* World Scientific Publishers, Singapore, 1993.

[150] M. Vardi. Verification of concurrent programs — the automata-theoretic framework. In *Proceedings of the 2nd IEEE Symposium on Logic in Computer Science*, pp. 167–176.

[151] T. Weigert and J. Tsai. A logic-based requirements language for the specification and analysis of real-time systems. In *Proceedings of the 2nd Conference on Object-Oriented Real-Time Dependable Systems*, Laguna Beach, 1996, pp. 8–16.