Automatically Verifying an Object-Oriented Specification of the Steam-Boiler System

Paulo J. F. Carreira [1] and Miguel E. F. Costa [1]

OBLOG Software S.A.
Al. António Sérgio 7, 1-A, 2795-023 Linda-a-Velha, Lisbon, Portugal

Abstract

Correctness is a desired property of industrial software systems. Although the employment of formal methods and their verification techniques in embedded real-time systems has started to be a common practice, the same cannot be said about object-oriented software. This paper presents an experiment of a technique for the automated verification of a subset of the object-oriented language OBLOG. In our setting, object-oriented models are automatically translated to LOTOS specifications using a programmable rule-based engine included in the Development Environment of the OBLOG language. The resulting specifications are then verified by model-checking using the CADP tool-box. To illustrate the concept we develop and verify an object-oriented specification of a well known case study—the Steam-Boiler Control System.

Key words: Automatic Verification, Code Generation, LOTOS, Model-Checking, Object-Oriented Systems, Steam-Boiler

1 Introduction

The employment of an automatic method for verifying properties about formal specifications known as *model-checking* [QS82, CES86, VW86, Kur90] experienced a dramatic growth. It has emerged as an effective way of finding errors and verifying correctness of hardware designs, and, more recently, software systems.

The applicability of this technique to software systems has been hindered by two different classes of problems. On the one hand, specifications of real-world

[1] E-mail: {pcarreira,ecosta}@oblog.pt, Tel.:+351-214146930, Fax:+351-214144125

systems often have state-spaces that are infinite or so large that their verification in an automated way is almost impossible in effective terms. Nevertheless, much effort has been put in additional techniques that, when used in a combined way, allow the exploration of the state-spaces of many real-world systems [CW96]. On the other hand, specification and verification techniques still require a degree of mathematical sophistication that make them inaccessible to the typical software engineer.

A promising compromise seems to lie in the combination of model-checking with the specification techniques that object-oriented graphical languages like UML Object Diagrams [BJR97] and StateCharts [HLN+90] have been proposing and advocating. However, producing complete specifications using such graphical languages is a labor-intensive task. These specifications often become overwhelming thus compromising the initial goal of being easier to read.

The object-oriented language OBLOG [OBL99] is being used in industry for the specification and deployment of critical parts of software systems [AS96]. OBLOG models can be developed by using both graphical and textual notations, making feasible the specification of complete systems with thousands of objects and classes.

In this paper we investigate the applicability of model-checking technology to the verification of object-oriented software specifications. We present an approach that allows fully automated verification by applying model-checking to Labelled Transition Systems (LTSs) that we derive from specifications in a subset of the OBLOG language.

Our approach is based on an intermediate translation from OBLOG to LOTOS [ISO88] specifications that are subsequently expanded to LTSs. This approach serves two important purposes. Firstly, it bridges the gap between the intuitive semantics of OBLOG and the needed formal semantics over LTSs. Indeed, because the language is still under development, there is no formal semantics yet for OBLOG against which the translation to LTSs can be proved to be sound. The fact that LOTOS is at a higher level of abstraction allows for this step to be much "smaller" and, hence, validated at an intuitive level. Secondly, the effort of implementing an algorithm to expand data non-determinism, made necessary by the very nature of object-oriented specifications, is greatly reduced by using Cæsar.Adt [Gar89], an abstract datatype compiler for LOTOS included in Cadp [FGK+96].

In order to test our ideas, we decided to work with a simplified version of the Steam-Boiler Control System, a well known example from the literature [ABL96], which allowed a faster analysis of the problem and provided other results for comparison.

Our paper is organized as follows: In Section 2, we present the requirements of a simplified version of the Steam-Boiler Control System and its modeling with OBLOG. The translation mechanism for producing LOTOS code is detailed in Section 3. We present and verify a formalization of the system requirements in Section 4, and Section 5 draws conclusions from this work.

1.1 Related Work

There have been other attempts to verify the Steam-Boiler System by model--checking but none of them, to the best of our knowledge, used a high level object-oriented language. In [WS96], Willig and Schieferdecker developed a Time-Extended LOTOS specification. The system was validated through simulation and verified for deadlock freedom using full state-space exploration techniques. They used Cadp on a restricted model without time and without failures.

A formalization of the problem into Promela without time is given by Duval and Cattel [DC96]. Their model also abstracts from communication failures and major properties of the system are reported to have been verified in a fully automated way using the Spin Model-Checker. Jansen et al. [JMMS98] report the verification of Amber specifications using a translation into Promela. This translation allowed the use of Spin in the automated verification of finite-state subsets of Amber.

2 Modeling the Steam-Boiler Controller System

The Steam-Boiler Control system is composed by a Micro-Controller connected to a physical system apparatus consisting of an Operator Desk and a Steam-Boiler attached to a turbine. There is also a Pump to provide water to the Boiler, an Escape Valve to evacuate water from the Boiler and devices for measuring the level of water inside the Boiler and the quantity of steam coming out.

The Boiler is characterized by physical limits M1 and M2, and a safety range between N1 and N2. When the system is operating, the water level can never go below M1 or above M2, otherwise the Boiler could be seriously damaged. The safety range establishes boundaries that, when reached, must cause a reaction from the Controller that reverts the increasing or decreasing tendency of the water level.

The Controller has different modes of operation, namely: *stopped*, *initialization*, *normal* and *emergency stop*. Initially the Steam-Boiler is switched off and the Controller is in stopped mode. System operations start when the start button of the operator desk is pressed. However, before the Boiler can start, the Controller must ensure that the water inside the Boiler is at an adequate level (between N1 and N2). To do this, it enters the initialization mode in which it uses the Water Pump and the Escape Valve to regulate the water level. When a safe range is reached, the Controller switches to normal mode and the production of steam initiates. In normal mode the Controller guarantees a safe water level inside the Boiler by starting and stopping the Pump. If something goes wrong, and the operator pushes the stop button, the Controller enters emergency stop mode and shuts down the Steam-Boiler.

The system can be further characterized by a set of requirements that are summarized as follows:

(1) When the start button is pressed and the system is stopped the Controller enters the initialization mode.
(2) When the Controller is in the initialization mode and the water level is below N1, the Pump must be started.
(3) When the Controller is in the initialization mode and the water level is above N2, the Valve must be opened.
(4) When the Controller is in the initialization mode and the water level is in the range N1 to N2, the Controller switches to normal mode.
(5) When the Controller switches to normal mode and the Valve is opened, the Valve must be closed.
(6) When the Controller is in normal mode, the Pump is started and the water level is above N2, the Pump must be stopped.
(7) When the Controller is in normal mode, the Pump is stopped and the water level is below N1, the Pump must be started.
(8) When the stop button is pressed the Controller enters emergency stop mode.
(9) When the water level of the Boiler is greater than N2, it will eventually become lesser than or equal to N2.
(10) When the water level of the Boiler is less than N1, it will eventually become greater than or equal to N1.
(11) If the Pump is started, the water will never reach a level above M2.
(12) If the Boiler is started, the water will never reach a level below M1.
(13) The Valve can only be opened if the Controller is in initialization mode.

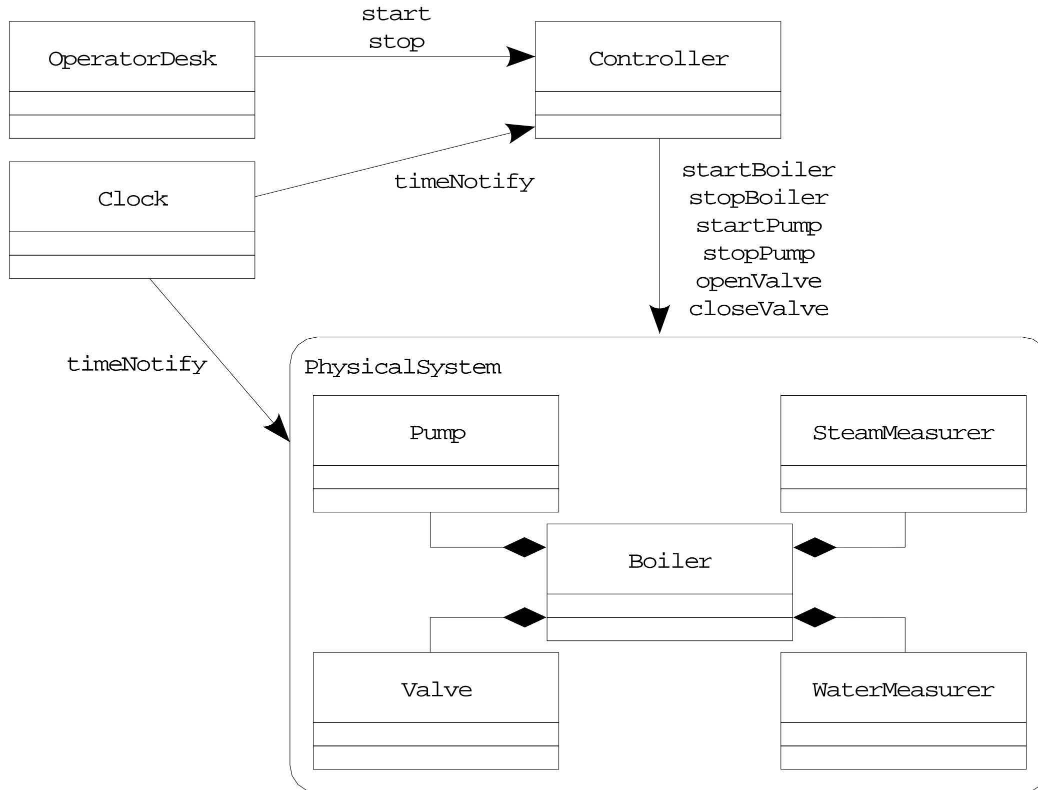

Fig. 1. Steam-Boiler Object Model

2.2 The OBLOG *Model*

OBLOG (OBject LOGic) refers both to a language and a development environment. The language OBLOG is a strongly-typed object-oriented specification language. Specifications are developed in a hierarchical fashion using *specification regions*. A specification region can be a class or an object encapsulating local declarations consisting of constants, attributes and operations as well as local specifications of datatypes and nested specification regions. Class and object operations can be implemented by several methods distinguished by corresponding enabling conditions.

In the original specification of the Steam-Boiler problem, the Controller interacts with the physical units through a single communication medium which has a specialized protocol defined for it. Our specification abstracts communication by modeling it with usual interaction between objects i.e., calls to object operations. However, we attempted to preserve the Controller's viewpoint by which the physical units are seen as a single entity composed by several other simpler entities.

At the top-level of our specification we have the `Controller` object, which

models the Controller software component, and the PhysicalSystem object, modeling the unified composition of all the physical units including the Steam-Boiler apparatus. In the specification region of this object are models of those units, namely the Boiler, Valve, Pump, WaterMeasurer and SteamMeasurer objects. Finally, also at top-level, are the OperatorDesk object and the Clock object, which is used to model time evolution.

In OBLOG there are two ways of initiating activity, *signal reaction* operations (denoted with a prefixing ^) and *self-fire* operations (denoted with a prefixing !). Reactions are triggered by signals sent by the external environment and we use them to model the events of pressing the start and stop buttons in the operator desk. Self-fire operations are used to model pro-active behavior. In our setting, because we do not have time constructs in OBLOG, time evolution was modeled with a self-fire operation of the Clock object named !clockTic().

The !clockTic() operation notifies both the PhysicalSystem and the Controller. The PhysicalSystem forwards this notification to the Boiler, which computes the new water level based on the current water level, the state of the Valve and Pump objects and its own internal state[2]. When the Controller is notified, it takes the appropriate actions according to its current operation mode as detailed above in the requirements section.

When a signal corresponding to the action of pressing the start or stop button is sent to the system, it is caught by the OperatorDesk object which contains two corresponding signal reaction operations named ^startButton() and ^stopButton() respectively. When the Controller is in stopped mode and the ^startButton() operation is triggered, the Controller is started. Similarly, when the ^stopButton() operation is triggered, the Controller is sent to emergency stop mode.

3 Translating OBLOG Specifications into LOTOS

OBLOG specifications can be automatically translated to given languages using an automatic code generation tool included in the OBLOG tool-set. Using this facility, we developed a translation of a sequential subset of the OBLOG language into LOTOS [ISO88], which is a standard Formal Description Language for software systems.

LOTOS is composed by two specialized sub-languages for specifying data and

[2] Recall that the Valve object can be either opened or closed, and both the Boiler and the Pump can be either started or stopped.

```
object Controller
declarations
  data types
    OperationMode = enum {
      Stopped, Initialization, Normal, Emergency} default Stopped;
  attributes
    mode : OperationMode := Stopped;
  operations
    start(); stop(); timeNotify();
body
  methods
    start method start is
      if (mode=Stopped) set mode:=Initialization; endif
    end
    timeNotify method tnStopped enabling mode=Stopped;
    end
    timeNotify method tnInit enabling mode=Initialization;
    local waterLevel:Integer;
    is
      call PhysicalSystem.getWaterLevel(waterLevel);
      if (waterLevel<N1) call PhysicalSystem.startPump(); endif
      if (waterLevel>N2) call PhysicalSystem.openValve(); endif
      if ((N1<=waterLevel) AND (waterLevel<=N2))
        call PhysicalSystem.closeValve();
        call PhysicalSystem.startBoiler();
        set mode:=Normal;
      endif
    end
    timeNotify method tnNormal enabling mode=Normal;
    local waterLevel:Integer;
    is
      call PhysicalSystem.getWaterLevel(waterLevel);
      if (waterLevel<N1) call PhysicalSystem.startPump(); endif
      if (waterLevel>N2) call PhysicalSystem.stopPump(); endif
    end
    timeNotify method tnEmergency enabling mode=Emergency;
    end
    stop method stop is
      if ((mode=Normal) OR (mode=Initialization))
        call PhysicalSystem.stopPump();
        call PhysicalSystem.closeValve();
        call PhysicalSystem.stopBoiler();
        set mode:=Emergency;
      endif
    end
end object
```

Fig. 2. Specification code of the Controller object

control parts. The data part is specified using the language ActOne [EM85] which is based on the theory of abstract datatypes. The control part is specified using a process algebraic language that combines and extends features of both CSP [Hoa85] and CCS [Mil89].

3.1 Translation Framework

The current framework is an evolution from previous studies in emulating subsets of the OBLOG language with process algebraic approaches to allow automatic verification [Car99]. These approaches are based on a translation that represents each object as a parallel composition of two recursively instantiated processes, one dedicated to the state and the other to the behavior of the object. The two processes synchronize through designated gates for reading and writing attribute values.

In fact, this coding relies heavily on LOTOS gates, also using them for both operation calls and parameter passing, resulting in a high degree of non-determinism which causes the explosion of the state-space. In our framework, in order to produce a LOTOS specification that can be compiled and verified in sensible time, an attempt was made to reduce non-determinism as much as possible; thus, gates were used as little as possible.

The state attributes of all the objects are merged into a global system variable that undergoes transformations corresponding to the behavior of the objects. To support this, special abstract datatypes are defined, namely a type *ObjType* that for each object Obj_i (i ranging in the number of objects in the system) with attributes $A_1 : T_{A_1}, \ldots, A_n : T_{A_n}$ defines a sort named $ObjSort_i$, and a type *SysState* that provides a representation of the global system state using each of the sorts $ObjSort_i$. The definition of the two datatypes is presented in Fig.3, where n is the number of attributes of object Obj_i and m is the number of objects in the system.

The main difference to the aforementioned approaches is that we do not use statements of the kind `G?s:SysState` in the LOTOS code, which are the main causes of the state-space explosion problem because they correspond to a non-deterministic choice ranging in the domain of the accepted variables. In fact, no part of the system state is explicitly sent through any gate. Rather, when operations are called, the corresponding processes that encode them are instantiated taking the system state as a parameter.

In OBLOG, an operation is composed by one or more *methods*, one of which is triggered, when that operation is called, according to enabling conditions that each has associated. A method, in OBLOG, can declare auxiliary local variables and its behavior is defined by an elementary action called *quark*. A

type $ObjState$ **is** $T_{A_1}, \ldots, T_{A_n}$
  sorts
    $ObjSort_i$
  constructors
    $mkObj_i : T_{A_1} \times \cdots \times T_{A_n} \rightarrow ObjSort_i$
  functions
    $setObj_iA_1 : ObjSort_i \times T_{A_1} \rightarrow ObjSort_i$
    $getObj_iA_1 : ObjSort_i \rightarrow T_{A_1}$
    $\ldots$
    $setObj_iA_n : ObjSort_i \times T_{A_n} \rightarrow ObjSort_i$
    $getObj_iA_n : ObjSort_i \rightarrow T_{A_n}$
  equations
  $\forall x_1 : T_{A_1}, \ldots, x_n : T_{A_n}$
  $\forall y_1 : T_{A_1}, \ldots, y_n : T_{A_n}$
    $setObj_iA_1(mkObj_i(x_1, \ldots, x_n), y_1) = mkObj_i(y_1, x_2, \ldots, x_n)$
    $getObj_iA_1(mkObj_i(x_1, \ldots, x_n)) = x_1$
    $\ldots$
    $setObjA_n(mkObj_i(x_1, \ldots, x_n), y_n) = mkObj_i(x_1, \ldots, x_{n-1}, y_n)$
    $getObj_iA_n(mkbj(x_1, \ldots, x_n)) = x_n$
endtype

type $SysState$ **is** $ObjType$
  sorts
    $SysState$
  constructors
    $mkSys : ObjSort_1 \times \cdots \times ObjSort_m \rightarrow SysState$
  functions
    $setObj_1 : SysState \times ObjSort_1 \rightarrow SysState$
    $getObj_1 : SysState \rightarrow ObjSort_1$
    $\ldots$
    $setObj_m : SysState \times ObjSort_m \rightarrow SysState$
    $getObj_m : SysState \rightarrow ObjSort_m$
  equations
  $\forall u_1 : ObjSort_1, \ldots, u_m : ObjSort_m$
  $\forall v_1 : ObjSort_1, \ldots, v_m : ObjSort_m$
    $setObj_1(mkSys(u_1, \ldots, u_m), v_1) = mkSys(v_1, u_2, \ldots, u_m)$
    $getObj_1(mkSys(u_1, \ldots, u_m)) = u_1$
    $\ldots$
    $setObj_m(mkSys(u_1, \ldots, u_m), v_m) = mkSys(u_1, \ldots, u_{m-1}, v_m)$
    $getObj_m(mkSys(u_1, \ldots, u_m)) = u_m$
endtype

Fig. 3. Abstract datatype specifications for types $ObjType$ and $SysState$.

quark can be a basic initiative (like assigning a value to an attribute or calling another operation) or it can be a sequential composition of other quarks. Our framework handles the composition of behavior (like operations composed by methods or quarks composed by sub-quarks) by translating the components

as subprocesses of the translation of the composite behavior.

In order to prevent the state-space explosion, another important issue is where activity starts. Instead of allowing any operation to be initiated at any time, activity initiates at only a few well-determined points in a single top-level recursive process, corresponding to the triggering of self-fire operations and reactions to external events. In each instantiation of this scheduler process, every enabled self-fire operation and every reaction to received external signals is called. In this context, the reception of signals is modeled as a choice between receiving or not receiving them i.e., calling the corresponding reaction operations or not.

On the first instantiation of the scheduler, the system state is initialized with the default values specified in the declaration of the objects. If an attribute of an object was not given a default value, we stipulate that the corresponding initial value is non-deterministically chosen in the range of the domain of that attribute. While not affecting the semantic mapping, this convention allows us to verify our properties for every possible initial scenario, in our case in particular, for every possibility of the water level inside the boiler at start-up.

3.2 Translation of Behavior Components

Generally, a behavior component *bc* (that can be an operation, a method or a quark) is translated to a process that receives the system state as a parameter, forwards it to the subprocesses or applies a transformation to it, returning a potentially altered version of the system state. The translation of *bc*, denoted by $\boldsymbol{proc}_{bc}$, renders the following:

```
proc_bc ≡
  process name_bc[G](s:SysState,in_bc) :
  exit(SysState, out_bc,Bool) :=
      action_bc
  where
      subprocs_bc
  endproc
```

where $\boldsymbol{G}$ is a set of gates, $\boldsymbol{name}_{bc}$ is a unique identifier for the behavior component, $\boldsymbol{action}_{bc}$ is the action taken by the behavior component and $\boldsymbol{subprocs}_{bc}$ is the declaration of subprocesses in the case of a compound behavior component. If *bc* is an operation with input (resp. output) parameters, these will be included in the $\boldsymbol{in}_{bc}$ (resp. $\boldsymbol{out}_{bc}$) list. Moreover, if *bc* is a method with local variables or a quark within a method with local variables, these will also be in $\boldsymbol{in}_{bc}$.

The execution of an OBLOG behavior component may result in failure in which case the `Bool` exit value of its corresponding LOTOS process is `true`. This is, however, not relevant in this report because the model we present does not allow failure in any case. This feature was only included in the framework for the sake of genericity.

3.2.1 Operations

If bc is an operation that has input parameters $I_1 : T_{I_1}, \ldots, I_n : T_{I_n}$, output parameters $O_1 : T_{O_1}, \ldots, O_m : T_{O_m}$ and is composed by methods $M_1, \ldots, M_n$, which have associated enabling conditions $\varepsilon_{M_1}, \ldots, \varepsilon_{M_n}$ respectively, we have:

$\boldsymbol{action}_{bc} \equiv$
  $[\varepsilon_{M_1}]$ `->` $\boldsymbol{name}_{M_1}$`(s,`I_1`,...,`I_n`)`
  `[]`
  $\cdots$
  `[]`
  $[\varepsilon_{M_n}]$ `->` $\boldsymbol{name}_{M_n}$`(s,`I_1`,...,`I_n`)`

$\boldsymbol{subprocs}_{bc} \equiv$
  $\boldsymbol{proc}_{M_1} \cdots \boldsymbol{proc}_{M_n}$

$\boldsymbol{in}_{bc} \equiv I_1 : T_{I_1}, \ldots, I_n : T_{I_n}$

$\boldsymbol{out}_{bc} \equiv T_{O_1}, \ldots, T_{O_m}$

3.2.2 Methods

If bc is a method such that: (1) its parent operation has inputs $I_1 : T_{I_1}, \ldots, I_n : T_{I_n}$ and outputs $O_1 : T_{O_1}, \ldots, O_m : T_{O_m}$ with default values $D_{O_1}, \ldots, D_{O_m}$; (2) has local variables $L_1 : T_{L_1}, \ldots, L_k : T_{L_k}$ with default values $D_{L_1}, \ldots, D_{L_k}$; and (3) Q is its implementation quark; we have:

$\boldsymbol{action}_{bc} \equiv$
  $\boldsymbol{name}_Q$`(s,`I_1`,...,`I_n`,`D_{O_1}`,...,`D_{O_m}`,`D_{L_1}`,...,`D_{L_k}`)`
  `>> accept s2:SysState,`
    $I'_1 : T_{I_1}, \ldots, I'_n : T_{I_n}, O_1 : T_{O_1}, \ldots, O_m : T_{O_m}$,
    $L'_1 : T_{L_1}, \ldots, L'_k : T_{L_k}$`,f:Bool`
  `in`
    `exit(s2,`O_1`,...,`O_m`,f)`

$\boldsymbol{subprocs}_{bc} \equiv \boldsymbol{proc}_Q$

$\boldsymbol{in}_{bc} \equiv I_1 : T_{I_1}, \ldots, I_n : T_{I_n}$

$\boldsymbol{out}_{bc} \equiv T_{O_1}, \ldots, T_{O_m}$

3.2.3 Quarks

In the context of a quark, no distinction is made between input parameters, output parameters and method local variables. Instead, if bc is a quark, we

say that it has a working set of variables declared as $V_1 : T_{V_1}, \ldots, V_n : T_{V_n}$ that subsume the previous declarations.

If bc is an operation call quark of the form **call** op(!I_1<<V_{I_1},...,!I_n<<V_{I_n}, !O_1>>V_{O_1},...,!O_m>>V_{O_m}) where !I_i<<V_{I_i} is an input binding associating input parameter I_i to a local variable V_{I_i}, and !O_i>>V_{O_i} is an output binding associating output parameter O_i to a variable V_{O_i}, we have that:

```
action_bc ≡
    name_op(s,V_I1,...,V_In)
    >> accept s2:SysState,
      O1:T_O1,...,Om:T_Om,
      f:Bool
    in
      exit(s2,V[Oi/V_Oi],f)
```

$\boldsymbol{in}_{bc} \equiv V_1 \!:\! T_{V_1}, \ldots, V_n \!:\! T_{V_n}$

$\boldsymbol{out}_{bc} \equiv T_{V_1}, \ldots, T_{V_n}$

where $\boldsymbol{V}$ represents the list of variables $V_1, \ldots, V_n$ and $\boldsymbol{V}[O_i/V_{O_i}]$ represents the list obtained from $\boldsymbol{V}$ by replacing each variable V_{O_i} with its corresponding bound value O_i.

To verify the system requirements, these will later be translated to formulas using predicates on the state of the objects. The generation procedure is parameterized with the predicates that belong to a particular formula. The obtained LOTOS specification is such that when modifying an object attribute, if the assignment causes any of these predicates to become true, an appropriate gate is signaled.

Let $p_1, \ldots, p_n$ be predicates that involve an attribute A that is modified and, for each p_i, let $p_i(s)$ designate the evaluation of the predicate in a given state s. The predicate checking procedure for attribute A is defined by the following processes, where i ranges in $1, \ldots, n$:

```
check_i ≡
    process checkP_i[gate_p1,...,gate_pn]
    (s1:SysState,s2:SysState) : exit :=
        [NOT(p_i(s1)) AND p_i(s2)] -> gate_pi ,
            checkP_i+1[gate_p1,...,gate_pn](s1,s2)
        []
        [p_i(s1) OR NOT(p_i(s2))] ->
            checkP_i+1[gate_p1,...,gate_pn](s1,s2)
    endproc

check_n+1 ≡
    process checkP_n+1[gate_p1,...,gate_pn]
    (s1:SysState,s2:SysState) : exit :=
        exit
    endproc
```

where s and s' represent the state of the system respectively before and after the modification of the attribute, and $\boldsymbol{gate}_{p_i}$ is the corresponding gate for each predicate p_i. If bc is an attribute modification quark of the form **set** A := exp where A is an attribute of an object Obj and exp is an expression of the same type as A, we have:

$\boldsymbol{action}_{bc} \equiv$
  checkP$_1$[$\boldsymbol{gate}_{p_1}, \ldots, \boldsymbol{gate}_{p_n}$](s,set*Obj*(s,set*A*(get*Obj*(s),*exp*)))
  >>
  exit(set*Obj*(s,set*A*(get*Obj*(s),*exp*)),***V***,false)

$\boldsymbol{subprocs}_{bc} \equiv \boldsymbol{check}_1 \cdots \boldsymbol{check}_{n+1}$

$\boldsymbol{in}_{bc} \equiv V_1 : T_{V_1}, \ldots, V_n : T_{V_n}$

$\boldsymbol{out}_{bc} \equiv T_{V_1}, \ldots, T_{V_n}$

The translation of other kinds of quarks, including the modification of local variables and the sequential and conditional quark compositions, is straightforward.

3.3 Automatic Generation

OBLOG language concepts are represented in an object-oriented Meta-Model as classes. An OBLOG repository can thus be regarded as a collection of instances of these classes.

The OBLOG Generator tool transforms repositories into actual implementations using transformation rules that map concepts described in the Meta-Model into constructs of a given target language. These transformation rules are written in RDL[OBL99] which is a “markup-like” scripting language executed in a specialized rule-execution engine.

An RDL rule executes under a given context which is an instance of the OBLOG Meta-Model. The implementation of a rule is a construct of the form `<$`*meta-class*`:`*rule-name*`>`$\cdots$`</$>` where *meta-class* is the (optional) declaration of the class of contexts (i.e., OBLOG Meta-Class) to which the rule can be applied (by default, a rule can be applied to any context). Preconditions can also be defined within the implementation of a rule to further constrain its application.

A context switch tag of the form `<@`*new-context*`>`$\cdots$`</@>` can be used in the implementation of a rule to explicitly alter the context of execution of the rule at that point. An iteration command of the form `<foreach` *collection*`>`$\cdots$`</foreach>` can be used to process collections. Within the `foreach`

```
<$ Quark:Action>::=
<pre Self.QuarkKind = CALL_QUARK>
  <@Self.CalledOp> <call Name> </@> '(s'
  <foreach Self.Variables>
    <before> ',' </before>
    <call UseVar> <sep> ',' </sep>
  </foreach> ')' <nl>
  '>> accept s2:SysState, '
  <foreach Self.CalledOp.OutputVars>
    <call DeclareVar> <sep> ',' </sep>
    <after> ',' </after>
  </foreach> 'f:Bool' <nl>
  'in' <nl>
  <tab> 'exit(s2,'
  <foreach SubstBind(Self.Variables, Self.CalledOp)>
    <call UseVar> <sep> ',' </sep>
    <after> ',' </after>
  </foreach> 'f)' <nl>
</$>
```

Fig. 4. Sample of RDL code that produces $\boldsymbol{action}_{bc}$ for call quarks.

command, the execution context is the current element of the collection being processed.

A rule can invoke the application of another rule through the `<call` *rule-name*`>` command. The invoked rule will inherit the current context at the point of the call command. Like a predicate in a logic program, a rule can have several implementations. Upon a call to a rule, one of its implementations is tried. If the context at the call point is not suitable for this implementation or if one of its preconditions fail another implementation is tried. If no implementations can be executed the failure is propagated to the calling rule.

The result of the application of a rule is either a transformation in a target repository or an output to a file. To write literal text to the currently open file inside a rule a string within quotation marks can be inserted at any point of the rule's implementation. To print the value of a variable (if allowed) the name of the variable, prefixed by a `$`, can also be inserted in the implementation of the rule.

4 Verification

Our ultimate goal is to verify that the Controller operates correctly i.e., that all the system requirements are guaranteed. A formal representation for each

of the requirements, given by a temporal logic formula, must be produced and verified.

To verify the formulas, we used the EVALUATOR Model-Checker included in the CADP tool-box [FGK+96]. CADP is a set of integrated tools for producing and analysing Labelled Transition Systems. LTSs can be obtained from low level descriptions, networks of communicating automata and high-level LOTOS specifications. Analysis functionalities include interactive simulation and verification through comparison of LTSs according to different simulation relations and model-checking.

4.1 Towards Formalization

First attempts at specifying the requirements in temporal logic yielded formulas that failed to verify because they did not exactly reflect the corresponding properties. In order to correctly check the model, these formulas required tuning. However, the huge size of the generated LTSs caused the Model-Checker to produce counter-examples that were cumbersome to understand for that purpose.

This prompted us to generate reduced versions of the LTSs, which allowed us to obtain smaller counter-examples by re-evaluating the formulas on the reduced graph. Also, we were now able to visually analyze those counter-examples using the graph drawing utility included in the CADP package.

The reduction process we used is based on Milner's observational equivalence relation of transition systems [Mil80], which preserves all sequences of visible actions. However, this reduction process does not preserve diverging paths in the original graph. As a consequence, some formulas that do not hold in a graph, verify successfully in its reduced version, meaning that they are divergence-sensitive and must be revised.

Generating graphs from the obtained LTSs can also be useful for debugging the OBLOG models. When modifications are made to a model, generating the graph can help in finding incorrect behavior, even before verifying any requirements. Fig. 5. shows a development process based on generating LTSs, obtaining corresponding graphs and verifying the LTSs through Model-Checking, which reflects the method we used to debug the Steam-Boiler OBLOG model and correctly specify the requirements in temporal logic.

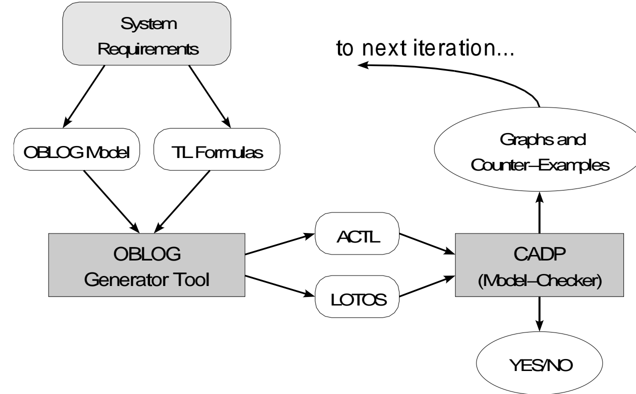

Fig. 5. Development process based on Model-Checking.

4.2 Requirements Formalization

A natural way of expressing properties about object-oriented systems is using a logic that allows one to express properties about states and actions, e.g., *when the Controller is in* ***stopped mode****, the valve will never* ***open***. In our setting, states are characterized by predicates like `Controller.mode` = *Stopped* and actions can be signal receptions like `^StartButtonPressed` or calls to object operations like `Valve.close()`. The ACTL (Action CTL) temporal logic [NV90] is appropriate for formalizing the Steam-Boiler requirements, being expressive enough for writing properties about states and actions. We selected a fragment of ACTL containing the following operators (besides usual logic connectors). Let p be a predicate, α a set of action labels and Φ an ACTL formula:

$$\Phi ::= p \mid \langle\alpha\rangle\Phi \mid [\alpha]\Phi \mid \mathbf{A}[\Phi_\alpha \mathbf{U} \Phi'] \mid \mathbf{A}[\Phi_\alpha \mathbf{U}_{\alpha'} \Phi']$$

Informally, the semantics of $\langle\alpha\rangle\Phi$ and $[\alpha]\Phi$ is that "eventually" (respectively "always") we reach states satisfying Φ performing "one" (respectively "all") actions denoted by α. The operator $\mathbf{A}[\Phi_\alpha \mathbf{U} \Phi']$ means that in all paths, Φ holds through α steps until it reaches Φ'. The operator $\mathbf{A}[\Phi_\alpha \mathbf{U}_{\alpha'} \Phi']$ means that in all paths, Φ holds through α steps until it reaches Φ' through an α' step. We write $\mathbf{AG}(\Phi)$ as a shorthand for $\mathbf{A}[\Phi_{true} \mathbf{U} false]$, meaning that all paths consist of states satisfying Φ.

As explained previously, the task of obtaining the temporal logic formulas to specify the requirements was simplified by the use of reduced graphical representations of the LTSs. For example, requirement 2, that states that *"when the Controller is in the initialization mode and the water level is below N1, the Pump must be started"*, could be specified as:

Fig. 6. Reduced graph obtained from a generated LTS.

$$\mathbf{AG}(\texttt{Controller.mode} = \mathit{Initialization} \land \texttt{Boiler.waterLevel} < N1 \Rightarrow \mathbf{A}[\mathit{true}_{\mathit{true}}\mathbf{U}_{\texttt{Pump.start()}}\mathit{true}])$$

Considering the graph[3] presented in Fig. 6., this formula is not a correct specification of the requirement because it does not cope with the fact that the stop button can be pressed before the pump is ever started. The correct formula would be:

$$\mathbf{AG}(\texttt{Controller.mode} = \mathit{Initialization} \land \texttt{Boiler.waterLevel} < N1 \Rightarrow \mathbf{A}[\mathit{true}_{\mathit{true}}\mathbf{U}_{\texttt{Pump.start()} \lor \texttt{\^{}StopButtonPressed}}\mathit{true}])$$

The system requirements can thus be formalized as:

(1) $\mathbf{AG}(\texttt{Controller.mode} = \mathit{Stopped} \Rightarrow [\texttt{\^{}StartButtonPressed}]\ \mathbf{A}[\mathit{true}_{\mathit{true}}\mathbf{U}\texttt{Controller.mode} = \mathit{Initialization}])$

(2) $\mathbf{AG}(\texttt{Controller.mode} = \mathit{Initialization} \land \texttt{Boiler.waterLevel} < N1 \Rightarrow \mathbf{A}[\mathit{true}_{\mathit{true}}\mathbf{U}_{\texttt{Pump.start()} \lor \texttt{\^{}StopButtonPressed}}\mathit{true}])$

(3) $\mathbf{AG}(\texttt{Controller.mode} = \mathit{Initialization} \land \texttt{Boiler.waterLevel} > N2 \Rightarrow \mathbf{A}[\mathit{true}_{\mathit{true}}\mathbf{U}_{\texttt{Valve.open()} \lor \texttt{\^{}StopButtonPressed}}\mathit{true}])$

[3] For simplicity, this graph was generated from a model that has the water level initialized at 0. This, however, does not affect the conclusions drawn from it.

(4) $\mathbf{AG}(\texttt{Controller.mode} = \mathit{Initialization} \wedge N1 \leqslant \texttt{Boiler.waterLevel} \leqslant N2 \Rightarrow$
$\mathbf{A}[\mathit{true}_{\mathit{true}}\mathbf{U}(\texttt{Controller.mode} = \mathit{Normal} \vee$
$\langle \hat{}\texttt{StopButtonPressed}\rangle \mathit{true})])$

(5) $\mathbf{AG}(\texttt{Controller.mode} = \mathit{Initialization} \wedge$
$N1 \leqslant \texttt{Boiler.waterLevel} \leqslant N2 \wedge \texttt{Valve.state} = \mathit{ValveOpened} \Rightarrow$
$\mathbf{A}[\mathit{true}_{\mathit{true}}\mathbf{U}_{\texttt{Valve.close()}}\mathit{true}])$

(6) $\mathbf{AG}(\texttt{Controller.mode} = \mathit{Normal} \wedge \texttt{Pump.state} = \mathit{PumpStarted} \wedge$
$\texttt{Boiler.waterLevel} > N2 \Rightarrow$
$\mathbf{A}[\mathit{true}_{\mathit{true}}\mathbf{U}(\texttt{Pump.state} = \mathit{PumpClosed} \vee$
$\texttt{Controller.mode} = \mathit{Emergency})])$

(7) $\mathbf{AG}(\texttt{Controller.mode} = \mathit{Normal} \wedge \texttt{Pump.state} = \mathit{PumpStopped} \wedge$
$\texttt{Boiler.waterLevel} < N1 \Rightarrow$
$\mathbf{A}[\mathit{true}_{\mathit{true}}\mathbf{U}(\texttt{Pump.state} = \mathit{PumpStarted} \vee$
$\texttt{Controller.mode} = \mathit{Emergency})])$

(8) $\mathbf{AG}(\texttt{Controller.mode} = \mathit{Initialization} \vee \texttt{Controller.mode} = \mathit{Normal} \Rightarrow$
$[\hat{}\texttt{StopButtonPressed}]\mathbf{A}[\mathit{true}_{\mathit{true}}\mathbf{U}\texttt{Controller.mode} = \mathit{Stopped}])$

(9) $\mathbf{AG}(\texttt{Controller.mode} \neq \mathit{Stopped} \wedge \texttt{Boiler.waterLevel} > N2 \Rightarrow$
$\mathbf{A}[\mathit{true}_{\mathit{true}}\mathbf{U}(\texttt{Boiler.waterLevel} \leqslant N2 \vee$
$\texttt{Controller.mode} = \mathit{Emergency})])$

(10) $\mathbf{AG}(\texttt{Controller.mode} \neq \mathit{Stopped} \wedge \texttt{Boiler.waterLevel} < N1 \Rightarrow$
$\mathbf{A}[\mathit{true}_{\mathit{true}}\mathbf{U}(\texttt{Boiler.waterLevel} \geqslant N1 \vee$
$\texttt{Controller.mode} = \mathit{Emergency})])$

(11) $\mathbf{AG}(\neg(\texttt{Pump.state} = \mathit{PumpStarted} \wedge \texttt{Boiler.waterLevel} > M2))$

(12) $\mathbf{AG}(\neg(\texttt{Boiler.state} = \mathit{BoilerStarted} \wedge \texttt{Boiler.waterLevel} < M1))$

(13) $\mathbf{AG}(\neg(\texttt{Controller.mode} \neq \mathit{Initialization} \wedge \texttt{Valve.state} = \mathit{ValveOpened}))$

4.3 Requirements verification

The Model-Checker we used does not allow the evaluation of predicates, and observations on the system state had to be included as actions in the model. As mentioned before, the generated LOTOS code can be augmented with gates that are signaled when a given condition p becomes true. The subsequent LTSs will be likewise enriched with transitions, labelled α_p, that are taken when that predicate is verified. In view of this, we can reformulate the properties, to a form allowed by the Model-Checker, as follows:

(1) $\mathbf{AG}([\alpha_{cond1A}]\mathbf{A}[true_{true}\mathbf{U}[\hat{}\texttt{StartButtonPressed}]\mathbf{A}[true_{true}\mathbf{U}_{(\alpha_{cond1B})}true]])$

(2) $\mathbf{AG}([\alpha_{cond2}]\mathbf{A}[true_{true}\mathbf{U}_{(\texttt{Pump.start()}\vee\hat{}\texttt{StopButtonPressed})}true])$

(3) $\mathbf{AG}([\alpha_{cond3}]\mathbf{A}[true_{true}\mathbf{U}_{(\texttt{Valve.open()}\vee\hat{}\texttt{StopButtonPressed})}true])$

(4) $\mathbf{AG}([\alpha_{cond4A}]\mathbf{A}[true_{true}\mathbf{U}_{(\alpha_{cond4B})\vee(\hat{}\texttt{StopButtonPressed})}true])$

(5) $\mathbf{AG}([\alpha_{cond5}]\mathbf{A}[true_{true}\mathbf{U}_{(\texttt{Valve.close()})}true])$

(6) $\mathbf{AG}([\alpha_{cond6A}]\mathbf{A}[true_{true}\mathbf{U}_{(\alpha_{cond6B})}true])$

(7) $\mathbf{AG}([\alpha_{cond7A}]\mathbf{A}[true_{true}\mathbf{U}_{(\alpha_{cond7B})}true])$

(8) $\mathbf{AG}([\alpha_{cond8A}]\mathbf{A}[true_{true}\mathbf{U}[\hat{}\texttt{StopButtonPressed}]\mathbf{A}[true_{true}\mathbf{U}_{(\alpha_{cond8B})}true]])$

(9) $\mathbf{AG}([\alpha_{cond9A}]\mathbf{A}[true_{true}\mathbf{U}_{(\alpha_{cond9B}\vee\alpha_{cond9C})}true])$

(10) $\mathbf{AG}([\alpha_{cond10A}]\mathbf{A}[true_{true}\mathbf{U}_{(\alpha_{cond10B}\vee\alpha_{cond10C})}true])$

(11) $\mathbf{AG}(\neg\langle\alpha_{cond11}\rangle true)$

(12) $\mathbf{AG}(\neg\langle\alpha_{cond12}\rangle true)$

(13) $\mathbf{AG}(\neg\langle\alpha_{cond13}\rangle true)$

where:

$cond1A \equiv (\texttt{Controller.mode} = Stopped)$

$cond1B \equiv (\texttt{Controller.mode} = Initialization)$

$cond2 \equiv (\texttt{Controller.mode} = Initialization \wedge \texttt{Boiler.waterLevel} < N1)$

$cond3 \equiv (\texttt{Controller.mode} = Initialization \wedge \texttt{Boiler.waterLevel} > N2)$

$cond4A \equiv (\texttt{Controller.mode} = Initialization \wedge N1 \leqslant \texttt{Boiler.waterLevel} \leqslant N2)$

$cond4B \equiv (\texttt{Controller.mode} = Normal)$

$cond5 \equiv (\texttt{Controller.mode} = Initialization \wedge$
$N1 \leqslant \texttt{Boiler.waterLevel} \leqslant N2 \wedge \texttt{Valve.state} = ValveOpened)$

$cond6A \equiv (\texttt{Controller.mode} = Normal \wedge$
$\texttt{Pump.state} = PumpStarted \wedge \texttt{Boiler.waterLevel} > N2)$

$cond6B \equiv (\texttt{Pump.state} = PumpStopped \vee \texttt{Controller.mode} = Emergency)$

$cond7A \equiv (\texttt{Controller.mode} = Normal \wedge$
$\texttt{Pump.state} = PumpStopped \wedge \texttt{Boiler.waterLevel} < N1)$

$cond7B \equiv (\texttt{Pump.state} = PumpStarted \vee \texttt{Controller.mode} = Emergency)$

$cond8A \equiv (\texttt{Controller.mode} = Initialization \vee \texttt{Controller.mode} = Normal)$

$cond8B \equiv cond9C \equiv cond10C \equiv (\texttt{Controller.mode} = Emergency)$

$cond9A \equiv (\texttt{Controller.mode} \neq Stopped \wedge \texttt{Boiler.waterLevel} > N2)$

$cond9B \equiv (\texttt{Boiler.waterLevel} \leqslant N2)$

$cond10A \equiv (\texttt{Controller.mode} \neq Stopped \wedge \texttt{Boiler.waterLevel} < N1)$

$cond10B \equiv (\texttt{Boiler.waterLevel} \geqslant N1)$

$cond11 \equiv (\texttt{Pump.state} = PumpStarted \wedge \texttt{Boiler.waterLevel} > M2)$

$cond12 \equiv (\texttt{Boiler.state} = BoilerStarted \wedge \texttt{Boiler.waterLevel} < M3)$

$cond13 \equiv (\texttt{Controller.mode} \neq Initialization \wedge \texttt{Valve.state} = ValveOpened)$

The verification yielded the following results, using a CADP installation on a 500MHz Intel machine with 128Mb of RAM running the Linux operating system:

Requirement number	*Lines of* LOTOS *code*	LOTOS *compilation timings*	*Number of states*	*Number of transitions*	*Verific. time*
1	1762	00’53.27”	215191	221763	00’22.22”
2	1786	00’49.68”	159815	164725	00’16.32”
3	1786	00’50.02”	159765	164675	00’16.26”
4	1806	00’56.00”	251418	259112	00’28.39”
5	1808	00’51.71”	160687	165597	00’16.08”
6	1838	00’55.69”	252625	260346	00’28.28”
7	1838	00’57.15”	253569	261263	00’28.43”
8	1762	00’51.77”	216362	222904	00’22.42”
9	1832	00’54.56”	252911	260605	00’46.43”
10	1804	00’57.16”	252896	260590	00’41.33”
11	1774	00’52.67”	159029	163939	00’16.35”
12	1774	00’49.10”	159526	164436	00’15.80”
13	1763	00’47.35”	140117	144421	00’13.73”

Each requirement corresponded to the generation of a single LOTOS specification from an OBLOG source file with 548 lines of code. All specifications were compiled and verified with a restriction on the integer domain to a range between 0 and 50.

5 Conclusions

Writing specifications using a high-level object-oriented language can be highly desirable. Typically, in many problem domains, using them for writing specifications is much easier. This promotes their use by domain experts wanting to skip the mathematical background needed by traditional specification languages.

We have seen how to verify properties of a subset of object-oriented specifications in a completely automated way. Our approach is based on a translation to LOTOS, which allowed us to establish a verification framework for the OBLOG language taking advantage of existing verification tools.

In the formalization of the system requirements, expressing apparently simple properties resulted initially in complex specification patterns. This seems to confirm [DAC98] that formalization in temporal logic can be quite error prone, although this effort increased our understanding of the problem through the analysis of the counter-examples provided by the Model-Checker. Indeed, some errors in our model were found and corrected.

Concerning the overhead of using an intermediate language, it can be claimed that a direct translation from OBLOG to LTSs could avoid many undesired transitions resulting from the LOTOS compilation. This direct translation can be enhanced by connecting to the API provided with the Open/Cæsar environment for generation and on-the-fly exploration of LTSs. However, by analyzing the obtained LOTOS specifications as high level representations of LTSs, we were able to isolate sources of non-determinism and devise strategies to optimize our initial translation.

This work is a contribution to a broader project that aims to provide automated verification of OBLOG specifications. For the moment we are leaving out features like *dynamic creation of objects*, *dynamic references* and *exception handling* which can result in infinite state-spaces. To cope with this, we are planning to incorporate techniques based on *abstraction* [CGL94], in particular we are looking at recent developments in the combined use of abstraction and program analysis techniques [DHZ99, SS98].

A formal semantics document for OBLOG is currently being organized. It will allow us to extend the supported subset of specifications and verify the correctness of this translation framework.

References

[ABL96] J. Abrial, E. Bger, and H. Langmaack, editors. *Formal Methods for Industrial Applications – Specifying and Programming the Steam Boiler Control*, volume 1165 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[AS96] Luis F. A. Andrade and Amilcar Sernadas. Banking and Management Information System Automation. In *Proceedings of the 13th world congress of the International Federation of Automatic Control (San Francisco, USA)*, volume L, pages 113–136. Elsevier-Science, 1996.

[BJR97] Grady Booch, Ivar Jacobson, and James Rumbaugh. *The Unified Modeling Language User Guide*. Addison-Wesley, 1997.

[Car99] Paulo J. F. Carreira. Automatic Verification of OBLOG Specifications. Master's thesis, Faculdade de Ciências da Universidade de Lisboa, Departamento de Informática, 1700 Campo Grande - Lisboa, 1999.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications. volume 8(2) of *ACM Transactions on Programming Languages and systems*, pages 244–263. 1986.

[CGL94] E.M. Clarke, O. Grumberg, and D. E. Long. Model Checking and Abstraction. volume 16(5) of *ACM Transactions on Programming Languages and Systems*, pages 834–871. 1994.

[CW96] E. M. Clarke and J. M. Wing. Formal Methods: State of the Art and Future Directions. volume 28(4es) of *ACM Computing Surveys*. December 1996.

[DAC98] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property Specification Patterns for Finite-State Verification. In Mark Ardis, editor, *In Proceedings of FMSP '98, The Second Workshop on Formal Methods in Software Practice*, pages 7–15. ACM Press, March 1998.

[DC96] Gregory Duval and Thierry Cattel. Specifying and Verifying the Steam Boiler Problem with SPIN. In Jean-Raymond Abrial, Egon Bger, and Hans Langmaack, editors, *Formal Methods for Industrial Applications – Specifying and Programming the Steam Boiler Control*, volume 1165 of *Lecture Notes in Computer Science*, pages 203–217. Springer-Verlag, 1996.

[DHZ99] Matthew B. Dwyer, John Hatcliff, and Hongjun Zheng. Slicing Software for Model Construction. In *ACM SIGPLAN Partial Evaluation and Program Manipulation*. January 1999.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification*, volume I. Springer-Verlag, 1985.

[FGK+96] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Radu Mateescu, Laurent Mounier, and Mihaela Sighireanu. CADP

(CÆSAR/ALDEBARAN Development Package): A Protocol Validation and Verification Toolbox. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the 8th Conference on Computer-Aided Verification (New Brunswick, New Jersey, USA)*, volume 1102 of *LNCS*, pages 437–440. Springer-Verlag, August 1996.

[Gar89] Hubert Garavel. Compilation of LOTOS Abstract Data Types. In Son T. Vuong, editor, *Proceedings of the 2nd International Conference on Formal Description Techniques FORTE'89 (Vancouver B.C., Canada)*, pages 147–162. North-Holland, December 1989.

[HLN+90] David Harel, H. Lachover, A. Naamad, Amir Pnueli, M. Politi, R. Sherman, A. Shtul-Trauring, and M. Trakhtenbrot. STATEMATE: A Working Environment for the Development of Complex Reactive Systems. volume 4 of *IEEE Transactions on Software Engineering*, pages 403–414. 1990.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[ISO88] ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, September 1988.

[JMMS98] Wil Janssen, Radu Mateescu, Sjouke Mauw, and Jan Springintveld. Verifying Business Processes using SPIN. In *Proceedings of the 4th International SPIN Workshop (Paris, France)*, 1998.

[Kur90] Robert P. Kurshan. Analysis of Discrete Event Coordination. In W. P. de Rover J. W. de Bakker and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, volume 430 of *Lecture Notes in Computer Science*, pages 414–453, Berlin, 1990. Springer-Verlag.

[Mil80] Robin Milner. A calculus of communicating systems. volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[Mil89] Robin Milner. *Communication and Concurrency.* International Series in Computer Science. Prentice Hall, 1989.

[NV90] R. De Nicola and F. W. Vaandrager. Action versus State Based Logics for Transition Systems. volume 469 of *Lecture Notes in Computer Science*, pages 407–419. Springer-Verlag, 1990.

[OBL99] OBLOG. The OBLOG Technical Information. Technical report, OBLOG Software S.A., www.oblog.com/tech, 1999.

[QS82] J. P. Queille and J. Sifakis. Specification and Verification of Concurrent Systems in CAESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351, Berlin, 1982. Springer-Verlag.

[SS98] D. A. Schmidt and B. Steffen. Data-Flow Analysis as Model-Checking of Abstract Interpretations. In G. Levi, editor, *Proceed-*

ings of the 5th Static Analysis Symposium, volume 1165 of *Lecture Notes in Computer Science*, Pisa, Italy, September 1998. Springer-Verlag.

[VW86] Moshe Y. Vardi and Pierre Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *Logic in Computer Science. IEEE Computer Society Press*, pages 332–344. 1986.

[WS96] Andreas Willig and Ina Schieferdecker. Specifying and Verifying the Steam Boiler Control System with Time Extended LOTOS. In Jean-Raymond Abrial, Egon Bger, and Hans Langmaack, editors, *Formal Methods for Industrial Applications – Specifying and Programming the Steam Boiler Control*, volume 1165 of *Lecture Notes in Computer Science*, pages 473–492. Springer-Verlag, 1996.