Dynamic Analysis of SA/RT Models Using SPIN

Javier Tuya, José R. de Diego, Claudio A. de la Riva, José A. Corrales

Universidad de Oviedo, Campus de Viesques
E.T.S. de Ingenieros Industriales e Informáticos
Carretera de Castiello, s/n. 33394 GIJON
Telef: 98-518-2049, FAX: 98-518-2150
E-mail: tuya@etsiig.uniovi.es

Abstract

This paper presents the integrated use of the Spin Model Checker in conjunction with Structured Methods (SA/RT). The graphical model is translated into a Promela program in which we prove assertions about the desired behaviour of the system. We also provide support for modular verification, by separately verifying different components of the model and deducing the desired global properties from the previous verifications. The above approach is then illustrated using a steam boiler system as a case study.

1. Introduction [*]

Development of Reactive and Real Time Systems is a critical task that requires our best efforts and techniques in order to attain more robust and reliable systems. The use of formal methods and verification techniques can help us in this task. We can distinguish between two distinct approaches for verification: Theorem Proving and Model Checking, but, from a practical point of view (and with the medium software engineer in mind), the use of model checking using the appropriate tools such as Spin [Holzmann, 91] is simpler than that of theorem proving. Despite the state-explosion problems, the ability of model checkers to find bugs is impressive, making it easy to see the payoff from formal verification [Dill, 96].

One of the more important aspects to consider in the development of Real Time Systems is the use of adequate notations and tools [Craigen, 93], which must be easy to use and must to generate documentation easy to review. Thence, our effort is addressed towards the use of a dual language approach [Felder, 94], using graphical tools (to support the development of an operational specification) and model checkers (in order to prove properties for the operational model). [Felder, 94] cites a large number of references in which the dual language approach is used. Others, in which the operational specification is developed using well known graphical methods, can be found in [Day, 93], [Damm, 94], where Statecharts are used in combination with a BDD based model checker; in [Tuya, 94], [Tuya, 95] with Structured Methods and the SMV model checker; in [León, 93], [Alonso, 95] where Structured Methods are used on top of a kernel of Petri Nets that can be further used for verification purposes; or in [Leue, 95] where Message Sequence charts are used in combination with Spin.

In this paper, we will use Structured Methods to model the system. The remainder of this section gives an overview of the RSML semantics that we will use. In Section 2., we will show how we can translate the model into Promela source code and check it using the verifier generated by Spin. Our approach is based on a two step execution semantics (micro pass and macro pass) and allows us to perform modular verification. In the second part of our paper we present a case study (Section 3.) and show the use of Spin for verification (Section 4.).

---

[*] This work has been funded by the "II Plan de Investigacion del Principado de Asturias" under project AUTOVER (PB-TIC95-01).

1.1. SA/RT Methods

Among the graphical methods most commonly used in industry, two of the leading methods are SA/RT and Statecharts. SA/RT is a short name for Structured Analysis Methods with extensions for Real Time [Ward, 85] [Hatley, 87]. The model is represented as a hierarchical set of diagrams that includes data and control transformations (processes). Control transformations are specified using State Transition diagrams, and events are represented using Control Flows. The other graphical and state based paradigm for specification of real time systems is Statecharts [Harel, 87a], [Harel, 90]. The system is represented as a set of hierarchical states instead of processes. Each state can be decomposed into sub states and so on. The statecharts notation is more compact than the SA/RT notation and has been formally defined. If this is so, why have we not selected Statecharts? The answer is simple:

- There are a lot of tools that support Structured Methods (both for Information Systems and Real Time), ranging from low to high cost, and these are well known by engineers. In contrast, there exist few (and expensive) tools for Statecharts.
- There is a lack of verification tools for Structured Methods, reduced in practice to syntax checking and completeness checking (e.g. balancing). The possibility to perform semantic checks will be a great added value to these tools.

1.2. Step Semantics

Before carrying out any verification effort, a clear execution semantics must be defined for the model. The original (informal) definition provided by Ward is inspired by the execution rules of Petri Nets. The execution of the model consists of a series of execution steps (external interactions). At each one, the environment can evolve, by placing some token(s) into flows. The system responds by executing some transformation(s) that place tokens into output flows and removing the input tokens. If some tokens are placed in control flows that can activate other transitions, the interaction continues until no transition can be executed. Then, other execution steps can start. This execution semantics has a high degree of non-determinism, because when two (or more) tokens are simultaneously placed, two (or more) branches of internal interactions must be executed sequentially, but in an arbitrary order. Each one must be completed before returning to the next [Ward, 86].

Another approach is based on the concept of macro-step and micro-step in Statecharts (their equivalent in Ward semantics as described above are external and internal interactions) [Harel, 87b]. At each macro-step, the environment changes and the system responds to it in a synchronous way, i.e. all inputs are taken simultaneously, the system responds in zero time and sets all outputs simultaneously. Each macro-step is composed of a chain of internal steps (micro-steps), in which a set of transitions are selected and executed in parallel. The new configuration can cause the execution of other transitions until no transition can be made. The essential difference with regard to Ward's approach is that more than one transition can be executed in parallel.

Although several different semantics have been defined for Statecharts (always based on the concept of micro and macro-steps) [Huizing, 91], we have adopted the more recent one, used in the RSML language that can be found in [Leveson, 94]. This is outlined below with a somewhat different notation.

Let *EI* be the set of External Input Flows, *EO* the set of External Output Flows, *F* the set of Internal Flows, and *S* the set of variables that defines the local state for each process. We denote by EI_i, EO_i, F_i, S_i the values of these variables at each step. The configuration C_i at step *i* is the set $EI_i \cup EO_i \cup F_i \cup S_i$. A macro-step starts due to a change in the environment variables that will have the set of values EI_0 and will give the new configuration C_0. In response, the system selects a set of transitions T_i to be triggered and executes them. It changes the current configuration from C_0 to C_1. The process repeats by triggering other transitions based on the previous configuration until no more transitions can be carried out. This execution is outlined below:

```
do (each Macro Step)
   i=0
   Changes to the Environment (new configuration is $C_0$)
   do (each Micro Step)
      Select $T_i$ for current configuration $C_i$
      Execute Transitions in $T_i$ (new configuration is $C_{i+1}$)
      i=i+1
   until $T=\emptyset$
end do
```

2. Using Spin and Promela for Verification

Promela provides a powerful mechanism for representing concurrent processes and the communication between them. But when using the RSML semantics, we must implement the simultaneous execution of several transitions and differentiate between micro and macro-steps. A direct translation from processes in the graphical model into processes in Promela will need a tight coordination between them. So, it is easier to implement the whole model as an unique process and the sequences of steps as loops.

2.1. Micro/Macro Step Semantics Implementation with Promela

The preliminary layout of the Promela program that implements this semantics will be:

```
do
   Changes to the Environment                              1
   do
      Select transitions T to be executed                  2
      Reset Input Flows                                    3
      if :: $T \neq \emptyset$ -> Execute Transitions      4
         :: else -> goto end_macro_step
      fi
   od
   end_macro_step:
   Assert Desired Properties                               5
   Reset Output Flows
od
```

1. External input flows (*EI*) can change their values. Since our desire is to prove properties that must be true for any execution of the system, these values are non-deterministically selected. This is a more realistic (and difficult to verify) assumption than the ones that suppose some kind of model for the environment behaviour.
2. Because the Spin structure is sequential, we cannot execute a transition that modifies the output flows because its result could influence another transition at the same micro-step. So, our first step is to select the set of transitions to be executed. A hidden variable is used to determine if a transition will be executed.
3. Before executing selected transitions *T*, we set all input control flows in $F \cup EI$ to zero. This situation will be needed as a prerequisite of the next micro-step.
4. When a transition can be executed, we continue by executing it (performing the actions that modify *EO*, *F* and *S*)
5. When no transition can be executed, we have finished the macro-step, The resulting configuration C_i represents the action of the whole macro-step, and we can check the assertions about the desired properties.

2.2. A More Refined Layout

The above layout is intuitive, but not practical when implemented in Promela. It must be modified by considering the following observations:

- It generates a high number of states that are not relevant for our verification purposes. We wish to enclose all sentences that are executed in a macro-step in an `atomic` sentence. So, the number of stored states will be the same of the observable states at the end of a step.
- If we put a loop inside the `atomic` sentence, there will be a potential infinite loop that can cause the verifier to hang. This problem will be even greater when we further include the concept of ghost variables. So, the specification must be restructured in such a way that all sentences in a micro-step will be executed in an atomic sentence, and so, a macro-step will be composed of one or more atomic steps. We differentiate between micro and macro-steps using a variable named `MicroStep`.
- The properties we wish to prove will be stated using assertions about current values of variables at the end of each macro-step. Typical assertions must ensure response or safety properties that relate external input flows to external output flows and to states. Sometimes we need to make assertions about previous values of variables and about internal flows. So as, to be able to do this, we define the following simple predicates:

  a *WAS0(f)*, where $f \in EI \cup F$: Is true if *f* was true at some instant in some preceding micro-step of the current macro-step.

  b *WAS1(f)*, were $f \in EI \cup EO \cup F$: Is true if *f* was true at some instant of the preceeding macro-step.

  c *INTERVAL(f_1 , f_2)*, where f_1 , $f_2 \in EI \cup EO \cup F$: Is set to true when f_1 becomes true, and changed to false when f_2 becomes false.
- Data transformations must be treated differently to control transformations, because they can execute some user supplied code. Data flows are discretized to relevant values, having a special value *NotPresent* that determines its presence or absence. The code for a discrete transformation will be executed when data becomes available. A continuous data transformation has an associated variable that can have one of the following values: *Disabled*, *Enabled*, *Scheduled*. At the start of each macro step all enabled continuous transformations are set to *Scheduled*, and so will be executed at the first micro-step. Execution leads the value of this variable to *Enabled*. If as a consequence of the execution of a transition a Data transformation is enabled, its value is set to *Scheduled* and hence, executed at the next micro-step.

Taking into account the above considerations, the new schema for our Promela program will be:

```
do
  atomic {
    if :: MicroStep==0 ->
         Changes to the Environment
         Calculate Predicates on EI
       :: else -> skip
    fi;
    Select Transitions to be Executed
    Execute Data Transformations (if any)
    Reset Input Flows
    if :: Some Transition will be Executed  ->
             Execute Transitions
             Calculate predicates on F
             MicroStep=1
          :: else ->
             Assert Desired Properties
             Calculate Predicates on EO
             Reset Output Flows
             MicroStep=0
    fi;
  }
od
```

2.3. Support for Modular Verification

Verification of realistically sized systems is still an open problem due to the state-explosion problem. Although Spin is very efficient, and enclosing each micro-step into an atomic sentence drastically reduces the set of reachable states, the problem still exists. Therefore, we will adopt a modular approach (also named rely/guarantee or assumption/commitment), in which properties will be stated for part of the model, and using the Abadi/Lamport composition theorem [Abadi, 93] we will deduce the truth of the property for the whole model. This theorem is based on the following induction rule:

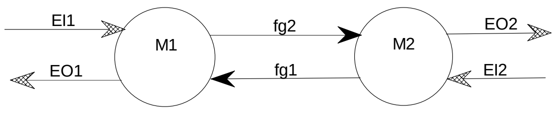
$$\frac{\begin{array}{c} M_1 \vDash_{\phi_{E1}} \phi_1 \,, \quad M_2 \vDash_{\phi_{E2}} \phi_2 \\ M_1 \vDash \phi_{E_2} \,, \quad M_2 \vDash \phi_{E_1} \end{array}}{[M_1 \| M_2] \vDash (\phi_1 \wedge \phi_2)}$$

Let M be the whole model, that can be decomposed into the parallel composition of $[M_1 \| M_2]$, and ϕ the desired safety property that can be decomposed as $\phi=\phi_1 \wedge \phi_2$. In order to prove ϕ over M, it is sufficient to prove:

- Each ϕ_i is true for the components M_i under the assumption ϕ_{Ei} for the environment of each M_i,
- Each component M_i grants the assumption ϕ_{Ej}, $i \neq j$ of each component over its environment (commitment).

Let us take a simple example represented in Figure 1. We wish to prove a property ϕ that relates EI_1 and EO_2. The model is divided into two components M_1 and M_2 communicated by internal flows fg_1, fg_2. The property ϕ is decomposed into two (both for M_1 and M_2 respectively). When we prove the property for M_2 we must assume some behaviour for fg_2.

Fig. 1: Modular Verification



Internal variables (fg_1 and fg_2 in the example) that communicate components which will be separately verified will be named ghost variables. When the component M_2 is translated into Promela, we give its ghost variables (fg_2) random values. Since in the whole model they are internal variables, their values can change at any instant of any micro-step and the set of sentences that give values to ghost variables will be placed inside the corresponding portion of code.

Assumptions restrict the behaviour of ghost variables, and so, we use a simple statement like:

ASSUME(cond,variable,value)

that must be read: If *cond* is true, then *variable* will be set to *value*. The following Promela code is placed immediately after the ghost variables have taken a value:

```
if :: cond -> variable=value
   :: else -> skip fi;
```

Assumptions are reversible: when we discharge the assumption (i.e. prove the committment for M_1), *ASSUME* sentences are coded as *COMMIT* sentences:

COMMIT(cond,variable,value)

that are then translated into a Promela assertion $cond \rightarrow \; variable=value$:

```
assert( ! cond || variable==value )
```

COMMIT sentences are translated into Promela after the end of each micro-step, as opposed to model properties, which are checked only at the end of each macro-step

By taking all the above considerations into account, the complete body of the Spin program will be:

```
Declare Variables and Macros
bit MicroStep==0
active proctype SART()
do
   atomic {
      if :: MicroStep==0 ->
progress:
          Changes to the Environment
          Calculate Predicates on EI
           :: else -> skip
      fi;
      Select Transitions to be Executed
      Execute Data Transformations (if any)
      Reset Input Flows
      Set Ghost Variables
      Make Assumptions
      if :: Some Transition will be Executed ->
               Execute Transitions
               Assert Commitments
               Calculate predicates on F
               MicroStep=1
            :: else ->
               Assert Desired Properties
               Calculate Predicates on EO
               Reset Output Flows
               MicroStep=0
      fi;
   } /*end atomic */
od
```

2.4. Implementation Issues

The automatic verification is currently being done as part of the AUTOVER project. The graphical specification (taken from a commercial CASE tool) is extracted to a textual file that describes the full set of transitions and variables using some C-like macros. For instance, `EXTINPUT(e)` represents an external input, `INTOUTPUT(f)` represents an internal flow, `GHOST(f)` establishes that `f` is a ghost variable, `TRANS(orig,dest,cond,acc)` represents a transition, `ASSUME(cond,f,v)` represents an assumption, etc.

The body of the Promela program is the same for all models and follows the structure outlined above. Each line of it is named a section. At each section, and using standard preprocessor directives, the full model is processed, but using different definitions for the C macros. For instance, `TRANS` is defined as an assignment sentence that sets the value of one variable in *T* when included in the select transitions section, into a set of assignments for output flows when defined in the execute section, and into the declaration of a hidden variable (*T*) when included in the declarations section.

Assertions that are violated give us an execution trail that shows all the steps taken by the program until assertion is violated. As we are only interested in the changes of the configuration and not in the internal steps taken by the Spin program, we do not use the `-p` option, but timely print statements are placed in order to visualize the desired steps when following the trail (option `-t`). These print statements are coded as a macro that is defined as `skip` when we run the verifier and into `printf` when we wish to visualize the counter-example (a sample can be found in Section

4.2.). On some occasions, counter-examples are extremely long. Due to this, we generate the verifier using the `-DREDUCE` option, and execute it with the `-i` or `-I` flag.

The experiments that will be presented in the following sections have been carried out using a low cost workstation (DecStation 5000/240). The use of modular verification means that verifications can be performed interactively. All the experiments report a maximum of 6Mb of memory usage and are completed in a maximum of 30 seconds for the worst case.

3. Case Study

The case study that we shall use is the "Steam Boiler Problem", that was suggested to the participants of the Dagstuhl Meeting on "Methods for Semantics and Specification" [Dagstuhl, 95].

The system is composed of a steam-boiler with four pumps, each having its own pump controller for supervision purposes, a device to measure the quantity of water in the steam-boiler, another for measuring the quantity of steam that comes out, and the corresponding operator desk and message transmission system.

The model of the system is not trivial, because the requirements specially emphasize the fault tolerant behaviour. The system must continue working under the failure of some devices. In such a case, it must internally estimate the measures that cannot be read from the faulty devices. Also, the system must recognize internal failures such as measures outside the dynamic range of the steam-boiler, spontaneous changes of states, etc. A full explanation of the whole specification is beyond the goal of this paper. Please refer to [Dagstuhl, 95] for more details.

The following sections will describe the SA/RT model[*] (only the relevant parts that will be used for this verification case study: top level control, a pump and the water level measurement system).

3.1. Overview of the Model

A first cut diagram is shown in the Steam Boiler chart (Figure 2) where we divide the system into processes. Circles drawned with dashed lines represent control transformations (that will be specified using state transition diagrams). The other circles represent either primitive data processes, or processes that are further decomposed in a lower level diagram. Arrowed dashed lines represent events (control flows) and continuous lines data flows. The Steam Boiler chart components are explained below (Figures 3 to 5):

- *Init Mode* is devoted to the initialization phase and will not be used later in this paper.
- *Control Modes* maintains the top level operation modes (*Init*, *Run* and *Emergency Stop*). In *Run* mode, the system can be in three different modes (*Normal*, *Degraded* or *Rescue*) depending on the particular combinations of failures of the devices.
- *Control Level* performs the main actions by opening and closing the pumps depending on the water level. The other processes control the behaviour of each device (water and steam sensors, and pump), each one having its own diagram.

[*] We have used a slight modification to the original SA/RT syntax that consists in the possibility of representing conditions as composed of logical expressions on control flows and states of other Processes. We denote the Enable, Disable and Trigger Activators as *X.Process* where *X* is *E*, *D* or *T*, and *Process* is the one that will be activated, deactivated or triggered. *S.Process* is also used to check the internal state of *Process*.

Fig. 2: Steam Boiler

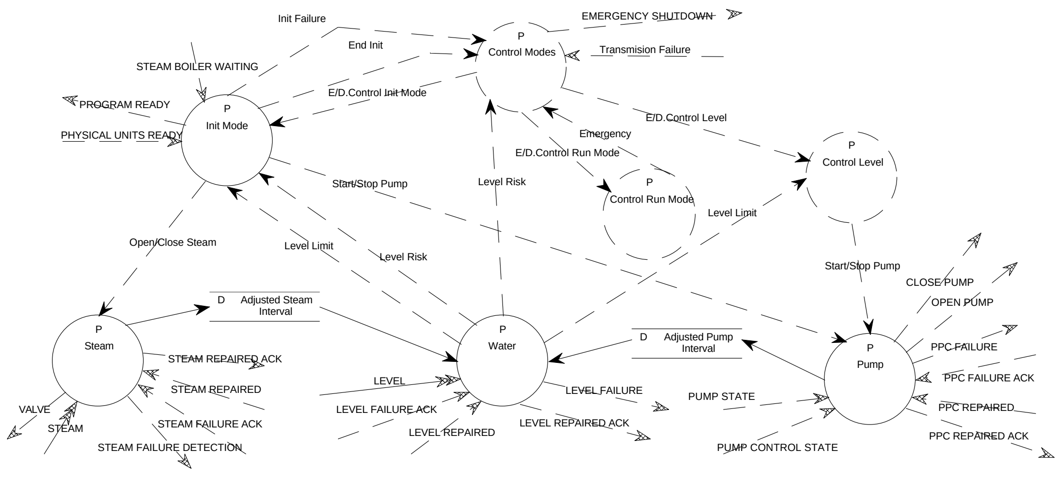


Fig. 3: Control Modes

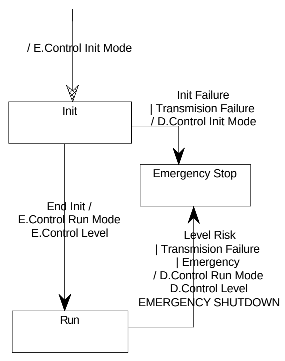


Fig. 4: Control Run Mode

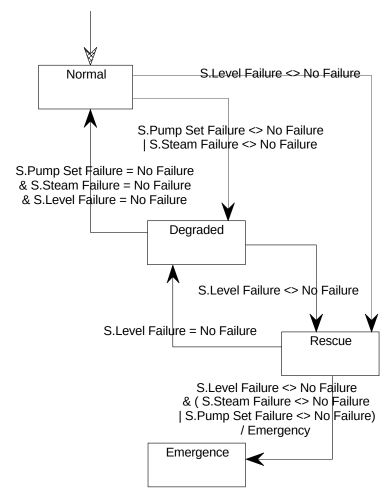
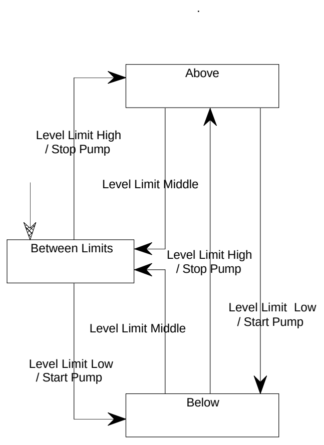

Normal
S.Level Failure <> No Failure
S.Pump Set Failure <> No Failure
| S.Steam Failure <> No Failure
S.Pump Set Failure = No Failure
& S.Steam Failure = No Failure
& S.Level Failure = No Failure
Degraded
S.Level Failure <> No Failure
S.Level Failure = No Failure
Rescue
S.Level Failure <> No Failure
& (S.Steam Failure <> No Failure
| S.Pump Set Failure <> No Failure)
/ Emergency
Emergence

Fig. 5: Control Level



3.2. Pump

Figure 6 is the *Pump* chart, which is composed of the following control processes (Figures 7 to 12):

- *Pump Status* maintains the desired pump state that has been set from *Control Level*. *Pump Set Failure Status* maintains the failure/not failure status for the set of pump and pump controller
- Because pump and pump controllers have independent behaviour and independent sources of failures, two processes (*Pump Failure Status* and *Pump Control Failure Status*) monitor the faulty states.
- Two more processes (*Pump Opening* and *Pump Closing*) are needed to model the maneouvres of starting and stopping pumps, and internal failures detection.

Fig. 6: Pump

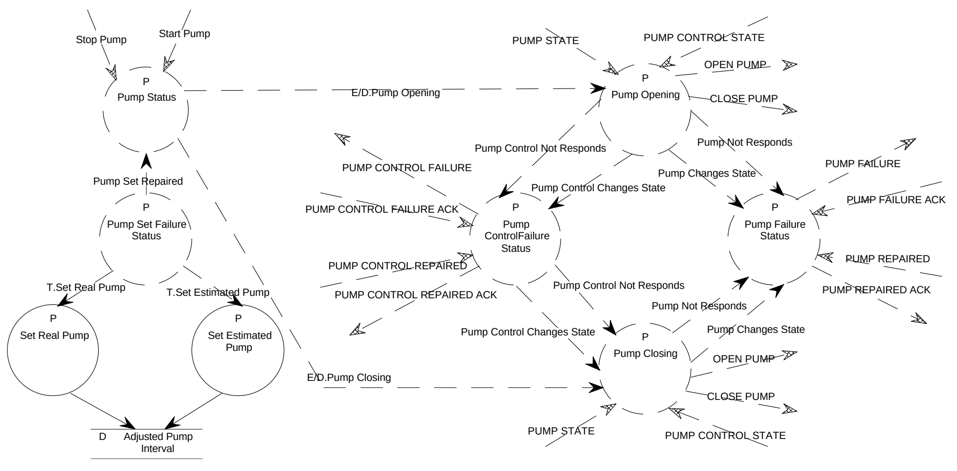


Fig. 7: Pump Status

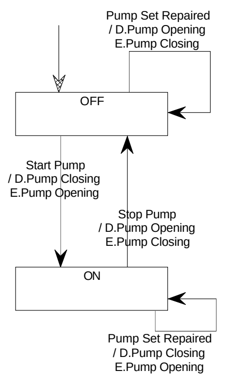

Pump Set Repaired / D.Pump Opening E.Pump Closing
OFF
Start Pump / D.Pump Closing E.Pump Opening
Stop Pump / D.Pump Opening E.Pump Closing
ON
Pump Set Repaired / D.Pump Closing E.Pump Opening

Fig. 8: Pump Failure Status

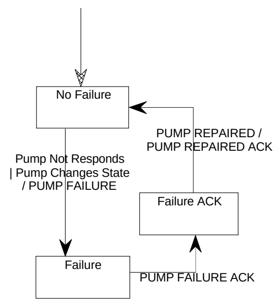

No Failure
PUMP REPAIRED / PUMP REPAIRED ACK
Pump Not Responds | Pump Changes State / PUMP FAILURE
Failure ACK
Failure
PUMP FAILURE ACK

Fig. 9: Pump Opening

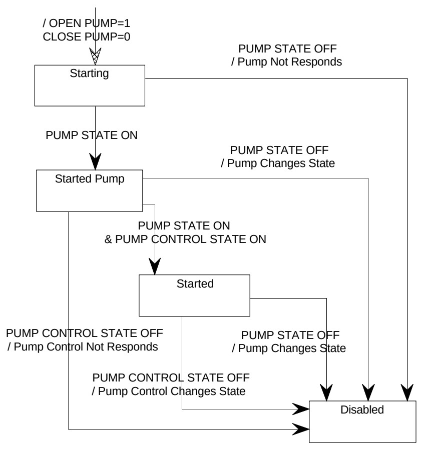

/ OPEN PUMP=1 CLOSE PUMP=0
Starting
PUMP STATE OFF / Pump Not Responds
PUMP STATE ON
PUMP STATE OFF / Pump Changes State
Started Pump
PUMP STATE ON & PUMP CONTROL STATE ON
Started
PUMP CONTROL STATE OFF / Pump Control Not Responds
PUMP STATE OFF / Pump Changes State
PUMP CONTROL STATE OFF / Pump Control Changes State
Disabled

Fig. 10: Pump Set Failure Status

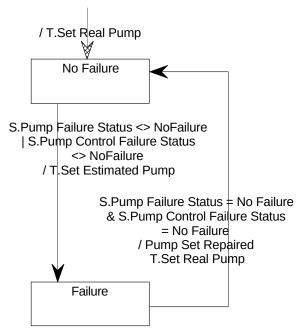

/ T.Set Real Pump
No Failure
S.Pump Failure Status <> NoFailure | S.Pump Control Failure Status <> NoFailure / T.Set Estimated Pump
S.Pump Failure Status = No Failure & S.Pump Control Failure Status = No Failure / Pump Set Repaired T.Set Real Pump
Failure

Fig. 11: Pump Control Failure Status

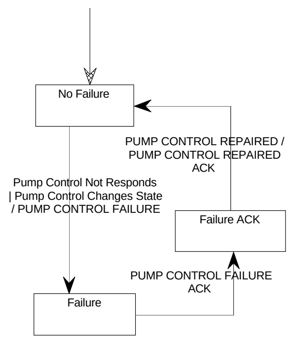
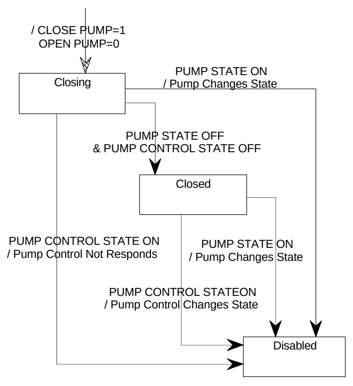

No Failure
PUMP CONTROL REPAIRED / PUMP CONTROL REPAIRED ACK
Pump Control Not Responds | Pump Control Changes State / PUMP CONTROL FAILURE
Failure ACK
PUMP CONTROL FAILURE ACK
Failure

Fig. 12: Pump Closing


/ CLOSE PUMP=1 OPEN PUMP=0
Closing
PUMP STATE ON / Pump Changes State
PUMP STATE OFF & PUMP CONTROL STATE OFF
Closed
PUMP CONTROL STATE ON / Pump Control Not Responds
PUMP STATE ON / Pump Changes State
PUMP CONTROL STATEON / Pump Control Changes State
Disabled

3.3. Water

Water is simpler than pumps. Its goal is to determine the quantity of water in the Steam Boiler and detect internal failures such as that the level measured is outside the dynamics of the system, or the absence of the *LEVEL* signal (see Figures 13 and 14).

Fig. 13: Water

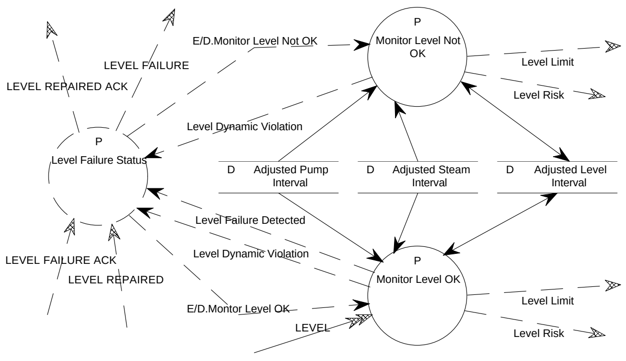
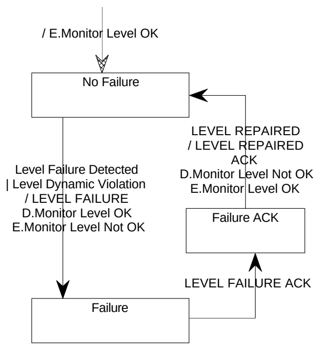


Fig. 14: Level Failure Status



/ E.Monitor Level OK
No Failure
LEVEL REPAIRED / LEVEL REPAIRED ACK D.Monitor Level Not OK E.Monitor Level OK
Level Failure Detected | Level Dynamic Violation / LEVEL FAILURE D.Monitor Level OK E.Monitor Level Not OK
Failure ACK
LEVEL FAILURE ACK
Failure

4. Verification

A large number of properties can be stated in order to safeguard the system described above. Part of them are explicitly included in the Dagstuhl Specification, and part of them can be deduced from it. We shall concentrate our effort on a small set of representative properties. The verification process will be explained as if we were incrementally developing and interactively debugging the model using the verifier.

4.1. Properties to be Verified

When the water level is low, the Pump must be open. If not, we must detect a failure and the operating mode must be different to *Normal*. The exact mode (*Degraded*, *Rescue* or *Emergence*) depends on the combination of failures for different devices. Because the pump does not respond immediately, we consider the Pump open if it is really open (*PUMP_STATE_ON*) or the Open command (*OPEN_PUMP*) has been sent. The formula that must be verified is:

$$\phi_1 : LEVEL=Low \rightarrow (OPEN\_PUMP \vee PUMP\_STATE\_ON \vee ControlRunMode \neq Normal)$$

Also, we consider the symmetric situation when the level is high:

$$\phi_2 : LEVEL=High \rightarrow (CLOSE\_PUMP \vee PUMP\_STATE\_OFF \vee ControlRunMode \neq Normal)$$

Modular verification will be conducted by partitioning the system into three components as shown in the following table:

Component	**Processes in the Structured Model**
MODES	*Control Modes , Control Run Mode*
WATER	*Water* (and its children), *Control Level*
PUMP	*Pump* (and its children)

Local formulas that we will prove are the following:

$$\phi_{P1} : \mathit{PUMP} \models \mathit{INTERVAL}(\mathit{StartPump},\mathit{StopPump}) \rightarrow (\mathit{OPEN\_PUMP} \vee \mathit{PUMP\_STATUS\_ON} \vee \mathit{PumpSetFailureStatus} \neq \mathit{NoFailure})$$

$$\phi_{P2} : \mathit{PUMP} \models \mathit{INTERVAL}(\mathit{StopPump},\mathit{StartPump}) \rightarrow (\mathit{CLOSE\_PUMP} \vee \mathit{PUMP\_STATUS\_OFF} \vee \mathit{PumpSetFailureStatus} \neq \mathit{NoFailure})$$

$$\phi_{L1} : \mathit{WATER} \models \mathit{LEVEL}=\mathit{Low} \rightarrow (\mathit{INTERVAL}(\mathit{StartPump},\mathit{StopPump}) \vee \mathit{LevelFailureStatus} \neq \mathit{NoFailure})$$

$$\phi_{L2} : \mathit{WATER} \models \mathit{LEVEL}=\mathit{High} \rightarrow (\mathit{INTERVAL}(\mathit{StopPump},\mathit{StartPump}) \vee \mathit{LevelFailureStatus} \neq \mathit{NoFailure})$$

$$\phi_{M} : \mathit{MODES} \models (\mathit{PumpSetFailureStatus} \neq \mathit{NoFailure} \vee \mathit{LevelFailureStatus} \neq \mathit{NoFailure}) \rightarrow \mathit{ControlRunMode} \neq \mathit{Normal}$$

It is clear that $\phi_{P1} \wedge \phi_{L1} \wedge \phi_{M} \rightarrow \phi_{1}$ and also $\phi_{P2} \wedge \phi_{L2} \wedge \phi_{M} \rightarrow \phi_{2}$. So, if we prove the above formulas we have proven our goals (ϕ_{1} and ϕ_{2}).

Because we observe that *WATER* and *PUMP* are closely related by means of the control flows *StartPump* and *StopPump*, a reasonable assumption for *PUMP* is that both events are never received simultaneously (in a micro-step). If we do not take this assumption into account now, we could be detect its need by examining the counter-examples produced in the PUMP verification. This assumption can be stated as $\neg(\mathit{StartPump} \wedge \mathit{StopPump})$ or as two implications:

$$(\mathit{StartPump} \rightarrow \mathit{StopPump}=0) \wedge (\mathit{StopPump} \rightarrow \mathit{StartPump}=0)$$

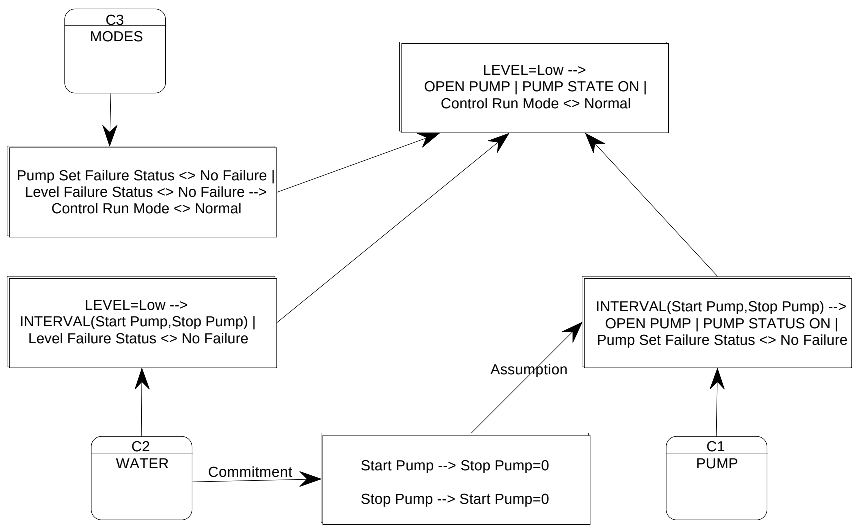
This will be represented using the macro ASSUME as follows:

$$\mathit{ASSUME}(\mathit{StartPump},\mathit{StopPump},0)$$

$$\mathit{ASSUME}(\mathit{StopPump},\mathit{StartPump},0)$$

The set of properties for ϕ_{1}, assumptions and commitments can be represented graphically as in the Figure 15, which is self-explanatory:

Fig. 15: Graphical representation of Properties for Modular Verification


C3
MODES
LEVEL=Low -->
OPEN PUMP | PUMP STATE ON |
Control Run Mode <> Normal
Pump Set Failure Status <> No Failure |
Level Failure Status <> No Failure -->
Control Run Mode <> Normal
LEVEL=Low -->
INTERVAL(Start Pump,Stop Pump) |
Level Failure Status <> No Failure
INTERVAL(Start Pump,Stop Pump) -->
OPEN PUMP | PUMP STATUS ON |
Pump Set Failure Status <> No Failure
Assumption
C2
WATER
Commitment
Start Pump --> Stop Pump=0
Stop Pump --> Start Pump=0
C1
PUMP

4.2. Verification of PUMP

First, we execute the Spin model checker for formula ϕ_{M1}, including the above assumptions and the variables *StartPump* and *StopPump* as ghost variables.

The counter-example (trail) produced shows that *PUMP* receives *StopPump* and closes the pump. Then, it fails. The last macro-step enables us to detect the problem (its corresponding part of the trail is included below).

```
*Macro Step: Environment
      **Received External Event PUMP_STATE_OFF
      **Received External Event PUMP_CONTROL_STATE_OFF
      **Received External Event PUMP_REPAIRED
  *Micro Step
     ** Transition  T41  : PumpFailureStatus  .  FailureACK  ->  NoFailure  ,
                    Action  PUMP_REPAIRED_ACK=1
  *Micro Step
     ** Ghost Event StartPump
     ** Transition  T11  : PumpSetFailureStatus .  Failure  ->  NoFailure ,
                    Action  PumpSetRepaired=1
  *Micro Step
     ** Transition  T01  : PumpStatus .  OFF  ->  OFF  ,
                    Action  PumpOpening=Disabled ; PumpClosing=Closing;
                    CLOSE_PUMP=1 ; OPEN_PUMP=0
  *Micro Step
    *No Actions. End Macro Step
spin: line 2913 "paper.i", Error: assertion violated
spin: trail ends after 4371 steps
```

In this last macro-step the event *PUMP_REPAIRED* is received from the environment. The internal sequence of micro-steps that will set the pump to the normal state starts. But *StartPump* is sent as a consequence of some (ghost) action on the environment in the second micro-step. In the third step, the process *PumpStatus*, which is in the *OFF* State, simultaneously receives the commands *PumpSetRepaired* and *StartPump*. Due to the non-determinism, the system goes to the *OFF* state and tries to close the pump by sending *CLOSE_PUMP*. Assertion is violated.

If we remove the non-determinism by giving priority to the transition from *OFF* to *ON*, the problem will be solved. The solution is to change conditions in the transition from *OFF* to itself to $PumpSetRepaired \wedge\neg StartPump$. Because the closing of the pump is symmetric, the same problem would appear for ϕ_{M2}, so we also change the transition from *ON* to itself to $PumpSetRepaired\wedge\neg StopPump$.

We run the verifier for the ϕ_{M1} formula and it is demonstrated as true. Since the closing property of the pump is symmetric, and the graphical specification seems to be symmetric, a naive analyst might assume that the closing properties ϕ_{M2} will also be true. But when we include them in the model checker we discover that they are false. The cause is that the problem is not truly symmetric because the *OFF* state in *PumpStatus* is the Initial state and the *ON* state is not. The brief counter-example shows us that while in the initial state, *PumpClosing* is received, but *PUMP_STATE_ON* indicates that the pump is open and there is no transition that can be executed in response to *PumpClosing*. Note that this problem does not appear while *PumpStatus* is *ON*, because the system goes to this state in response to the first *StartPump*. A failure, when repaired, causes the *PumpSetRepaired* event to be received, and so, the pump will be open again.

We solve the problem by changing the transition from *OFF* to *OFF* to obtain the condition $(PumpSetRepaired\wedge\neg StartPump)\vee StopPump$. The transition from *ON* to *ON* does not need any Change.

Additional properties which can be asserted and proven true of the desired response of the pump are:

$$WAS1(OPEN\_PUMP) \rightarrow PUMP\_STATE\_ON \vee PumpSetFailureStatus \neq NoFailure$$

$$WAS1(CLOSE\_PUMP) \rightarrow PUMP\_STATE\_OFF \vee PumpSetFailureStatus \neq NoFailure$$

or of the undesired response:

$$INTERVAL(StartPump,StopPump) \rightarrow CLOSE\_PUMP=0 \vee WAS0(StopPump)$$

$$INTERVAL(StopPump,StartPump) \rightarrow OPEN\_PUMP=0 \vee WAS0(StartPump)$$

4.3. Verification of WATER

We follow the same procedure for the *WATER* component. In this case, *StartPump* and *StopPump* will not be ghost variables, and the assumptions will be proven using the corresponding commitments

$$COMMIT(StartPump,StopPump,0)$$

$$COMMIT(StopPump,StartPump,0)$$

Recall that commitments are checked after each micro-step, as opposed to the system properties, which are checked at the end of each macro-step.

Formulas ϕ_{L1} and ϕ_{L2} are checked as true.

We can also state other interesting safety properties from the original specification. For instance, whenever *LEVEL* is detected to be outside the system dynamics, we consider the Level sensor to be failing, or if *LEVEL* is not present, the system detects a failure:

$$OutOfDynamics \rightarrow LevelFailureStatus \neq NoFailure$$

$$LEVEL=NotPresent \rightarrow LevelFailureStatus \neq NoFailure$$

4.4. Verification of MODES

We verify the formula ϕ_M for *MODES* and find it true. Other properties that have been stated in the original specification are the invariant for the operating modes:

$$ControlRunMode=Normal \rightarrow (PumpSetFailureStatus=NoFailure \wedge SteamFailure=NoFailure \wedge LevelFailure=NoFailure)$$

$$ControlRunMode=Degraded \rightarrow ((PumpSetFailureStatus \neq NoFailure \vee SteamFailure \neq NoFailure) \wedge LevelFailure=NoFailure)$$

$$ControlRunMode=Rescue \rightarrow (\neg(PumpSetFailureStatus \neq NoFailure \vee SteamFailure \neq NoFailure) \wedge LevelFailure \neq NoFailure)$$

that also are found true.

Finally, we can conclude that our goal formulas ϕ_1 and ϕ_2 are true over the whole model.

4.5. Searching for Non Progress Cycles

One of the problems of the RSML semantics is the possibility of falling into infinite loops in which the system executes an infinite sequence of micro-steps. This type of potential deadlock risk must be detected in order to safeguard performance. We must ensure that the system executes an infinite sequence of macro-steps. To do this, we have labeled the section of the Promela in which the environment changes its value with a progress label, and by means of the spin option `-l` we can check the progress. We repeat the verification of each model using this option and we detect more problems:

Spin detects that *PUMP* has a non progress cycle, since *StartPump* and *StopPump* can appear infinitely often in the same micro-step (recall that they are ghost variables). Although this might not be a problem if WATER does not exhibit this behaviour, it is a potential risk. The approach that we have taken is to restrict the behaviour of ghost variables by stating that they appear at most, only once in the same macro-step:

$$ASSUME \ (WAS0(StartPump) \vee WAS0(StopPump) \rightarrow StartPump=0)$$

$$ASSUME \ (WAS0(StopPump) \vee WAS0(StartPump) \rightarrow StopPump=0)$$

On running the verifier with the `-l` option we detect the absence of non progress cycles. Because these assumptions restrict the freedom of the PUMP behaviour if we run the verifier without the `-l` option, we prove that properties are also true.

Now, using *WATER*, we try to discharge these new assumptions by translating them into commitments, and we find that *WATER* does not guarantee them. The counter-example shows us that *LevelFailure* is in *Failure* state. *Monitor Level Not OK* data process can determine (in the first micro-step) the level limits that are received by *Control Level* which sends a Start/Stop command to the pump. But if *LEVEL REPAIRED* is received, Data Transformation *MonitorLevelOK* is enabled and another start/stop message is sent (in the second micro-step).

We must transform the assumption/commitment into another "softer" one:

$$ASSUME\ (\ WAS0(StartPump) \rightarrow StartPump=0)$$

$$ASSUME\ (\ WAS0(StopPump) \rightarrow StopPump=0)$$

We verify the above formulas both for *PUMP* as assumptions and for *WATER* as commitments, and re-check the existence of non progress cycles. Similar assumptions must be made for the *MODES* component by assuming that in the same micro-step, transitions from/to the *NoFailure* states are never taken more than once, along with the corresponding commitments for *PUMP* and *WATER*. Now the specification is found to be true.

5. Conclusions

We have shown how the Spin Model Checker can be used in conjunction with graphical specification methods, and how we can use it for debugging the specification as well as proving some interesting properties in said specification. The modular approach allows us to perform separate verifications for components of the model and compose them in order to prove properties for the whole model. The Promela program that represents the model is slightly more complicated and less straightforward than if it were coded directly from the textual specification as in [Duval, 95], but its great advantage is that it is automatically obtained from a graphical specification, and so, easier to develop, review and maintain.

Our present work, as part of the AUTOVER project, is addressed towards a complete integration of structured development tools and model checkers. The user specifies the system graphically and the results of the verification (counter-examples) can be graphically animated in order to debug the model. We also represent the whole set of properties to verify in a similar way to that shown in Figure 15. This allows us to perform regression verifications and detect the problems caused by changes in the model. We also plan to compare Spin with other model checkers [Corbett, 96], and to study the possibilities of automatic reductions of the model [Simone, 94] and relation checking [Erdogmus, 95], both for the part of model that is being verified as well as for its environment.

References

- Abadi, M., Lamport, L. (1993) "Conjoining Specifications". DEC Software Research Centre. Research Report no. 118.
- Alonso, A., Baresi, L., Christensen, H.,Heikkinen, M. (1995) “IDERS: An Integrated Environment for the Development of Hard Real-Time Systems”. Euromicro Workshop on Real-Time Systems. Odense, Denmark.
- Corbett, J. C. (1996) "Evaluating Deadlock Detection Methods for Concurrent Software". IEEE Transactions on Software Engineering, Vol. 22, no. 3, March, pp. 161-180.
- Craigen, D., Gerhart, S., Ralston, T. (1993) "An International Survey of Industrial Applications of Formal Methods" (2 Vols.). U.S. Department of Commerce, Technology Administation, National Institute of Standards and Technology, Computer Systems Laboratory Gaithersburg. Report NISTGCR 93/626.
- Damm, W., Hungar, H., Kelb, P., Schlör, R. (1994) "Using Graphical Specification Languages and Symbolic Model Checking in the Verification of a Production Cell". In Case Study "Production Cell". Claus Lewerentz anb Thomas Lindner Eds., pp. 89-107.

- Dagstuhl (1995). Seminar on "Methods for Semantics and Specification" Schloß Dagstuhl, Wadern, Germany, on June 5-9. Available in http://www.informatik.uni-kiel.de/~procos/dag9523/dag9523.html until publication.
- Day, N. (1993) "A Model Checker for Statecharts (Linking CASE Tools with Formal Methods)", MS Thesis, University of British Columbia, Dept. of Computer Science (Tech. Report 93-35).
- Dill, D. L., Rushby, J. (1996) "Acceptance of Formal Methods: Lessons from Hardware Design". IEEE Computer, vol. 29, no 4, April, pp. 23-24
- Duval, G., Cattel, T. (1995) "Specifying and Verifying the Steam Boiler problem with SPIN". In [Dagstuhl 95].
- Felder, M., Mandrioli, D., Morzenti, A. (1994) "Proving Properties of Real-Time Systems Through Logical Specifications and Petri Net Models". IEEE Transactions on Software engineering, Vol. 20, no. 2, Februray, pp. 127-141.
- Harel, D. (1987a) "STATECHARTS: A Visual Formalism for Complex Systems". Science of Computer Programming, North Holland, Vol. 8, pp. 231-274.
- Harel, D., Pnuelli, A., Pruzan-Schmidt, J., Sherman, R. (1987b) "On the formal semantics of Statecharts". Proceedings of the 2nd Symposium on Logic in Computer Science, pp. 56-64.
- Harel, D, Lachover, H., Naamad, A, Pnueli, A., Politi, M., Sherman, R., Tachtenbrot, M. (1990) "Statemate: a working environment for the development of complex reactive systems". IEEE Transactions on Software Engineering, Vol. 16, no. 4, pp. 403-414.
- Hatley, D. J. , Pirbhai, I. (1987) "Strategies for Real Time System Specification". Dover Press, New York.
- Herdogmus, H. (1995) "Verifying Semantic Relations in SPIN". Proceedings of the 1st SPIN Workshop. Montreal.
- Holzmann, G. J. (1991) “Design and Validation of Computer Protocols”. Prentice-Hall
- Huizing, G. (1991) "Semantics of Reactive Systems: Comparison and Full Abstraction". Ph.D. Thesis, Technische Universiteit of Eindoven.
- León, G., Dueñas, J. C., Puente, J. A., Alonso, A. (1993) "The IPTES Environment: Support for Incremental, Heterogeneus and Distributed Prototyping". Real-Time Systems Journal, pp. 153-172.
- Leue, S., Ladkin, P. B. (1995) "Implementing Message Sequence Charts in Promela". Proceedings of the 1st Spin Workshop. Montreal.
- Leveson, N. G., Heimdahl, M. P. E., Hildreth, H., Reese, J. D. (1994) "Requirements Specification for Process Control Systems" IEEE Transactions on Software Engineering, Vol. 20, no. 9, pp. 684-707.
- Simone, R., Ressouche, A. (1994) "Compositional Semantics of Esterel and Verification by Compositional Reductions". In: Computer Aided Verification, Springer LNCS-818.
- Tuya, J., (1994) "Specification and Verification of Reactive Systems Using Structured Methods and Temporal Logic". Ph.D. Thesis. Universidad de Oviedo.
- Tuya, J., Sánchez, L., Corrales, J. A. (1995) "Using a Symbolic Model Checker for Verify Safety Properties in SA/RT Models". 5th European Software Engineering Conference. Springer LNCS 989, pp. 59-75
- Ward, P., Mellor, S. (1985) "Structured Development for Real-Time Systems". Prentice-Hall.
- Ward, P. (1986) "The Transformation Schema: An Extension of the Data Flow Diagram to Represent Control and Timing". IEEE Transactions on Software Engineering, Vol. 12, no. 2, pp. 198-210.