LARES - A Novel Approach for Describing System Reconfigurability in Dependability Models of Fault-Tolerant Systems

Max Walter
Technische Universität München, Germany

Alexander Gouberman, Martin Riedl, Johann Schuster, Markus Siegle
Universität der Bundeswehr München, Germany

This paper presents LARES, a novel approach to the modeling of fault-tolerant systems. We introduce a formalism for describing the structure of a system which is able to express dynamic behavior such as imperfect coverage, common cause errors, failure propagation, increase of failure rates after partial system failure, and phased missions. It is designed with the intention to provide a convenient and easy-to-learn formalism for modeling – even for non-specialists. The paper introduces the modeling language and illustrates its use by means of two non-trivial examples.

1 INTRODUCTION

Due to the dynamic behavior of reconfigurable fault-tolerant systems, the creation of stochastic dependability models is a difficult task. Traditional techniques like fault trees or reliability block diagrams are no longer sufficient in many cases, because they assume all components to be of a Boolean nature. However, in today's adaptable and reconfigurable systems, components must be described by more than the states 'active' and 'failed' in order to reflect the different roles of a component in a reconfigurable system. Moreover, often the system itself is not considered to be Boolean, but different failure classes are discriminated. Finally, the basic events (component failures and repairs) cannot be assumed to be independent, but common cause failure, failure propagation, limited repair capacities etc. must be taken into account.

Therefore, methods like Markov chains, stochastic Petri nets or models based on a stochastic process algebra must be applied. However, these methods are usually considered to be too formal, error-prone, and work-intensive to be applicable in most industrial scenarios.

To increase the applicability of state-based methods, several approaches have been developed in order to automatically generate these models from high-level input specifications (see Sec. 2). All these approaches convert one specific type of input to one specific type of state-based model. In order to be more flexible, we suggest to use an intermediate model representation, which can be generated from different kind of application-specific input, and from which different type of state-based models and be generated. Hence, different types of analytical solvers or simulators readily available can be applied for quantitative evaluation. The intermediate model must neither be too low-level nor too high-level: a tradeoff must be found in order to be able to easily generate the intermediate model and to easily convert it into the state-based low-level representation. In this paper, we present a possible candidate for such a tradeoff called LARES (modeling LAnguage for REconfigurable Systems) . LARES supports both the textual and graphical representation of dependability models which allows for a manual editing of the automatically generated intermediate models. Models of larger systems are modularized and arranged in a hierarchic manner, allowing for efficient quantitative evaluation. Regarding modeling power, features of LARES include: non-Boolean systems and non-Boolean basic events, repairable and non repairable systems, reliability and availability analysis, distinction of hot-, warm- and cold-redundancy, fault-tolerant redundancy groups and system reconfiguration after failures, repairs, or other events.

The paper is organized as follows: Sec. 2 presents some related work. Sec. 3 describes the basic modeling formalism in textual form. In Sec. 4 we apply our specification formalism to two case studies from the literature showing the power of our new approach. Sec. 5 concludes the paper.

2 RELATED WORK

2.1 *SAVE*

The System AVailability Estimator (SAVE, (Goyal et al. 1986; Blum et al. 1994; Blum et al. 1993)) is one of the earliest tools which are able to automatically generate Markov Chains for dependability evaluation. The model is specified using a text file describing the components and the system's redundancy structure, the available repair resources and the repair strategies, and other inter-component dependencies. In comparison with more modern approaches, the modeling power of SAVE is limited. A graphical representation does not exist and there is no support for hierarchical models.

2.2 *Dynamic Fault Trees (DFT)*

Another tool which transforms high-level input diagrams into state-spaced models is Dynamic Innovative Fault Tree (DIFtree, (Dugan et al. 1992; Manian et al. 1998; Coppit & Sullivan 2000; Dugan et al. 2000; Dugan et al. 1997)). Dynamic fault trees extend traditional fault trees by a set of novel gates: PAND (priority AND) are special AND-gate which take into account the sequence in which the events occur. A SPARE-gate allows for modeling varying failure rates in cold- or warm-standby redundant systems. Finally, FDEP-gates are used to model deterministic failure propagation. To avoid disambiguities, dynamic fault tree gates have been formally defined using so-called interactive Markov chains. This also allows for a more efficient evaluation method of DFT which has been implemented in the tool CORAL (COmpositional Reliability and Availability anaLysis, (Boudali et al. 2007b; Boudali et al. 2007a)).

2.3 *Arcade*

Within the architectural dependability evaluation (Arcade) framework (Boudali et al. 2008), fault-tolerant systems are modeled using three different types of building blocks: basic components, repair units and spare management units. The semantics of these blocks and their interaction is defined using interactive Markov chains. The syntax of an Arcade model bears similarities to SAVE. However, due to the definition of the basic building blocks by interactive Markov chains, Arcade allows for compositional state space generation and reduction. Moreover, the modeling language can be extended more easily.

2.4 *Dynamic Reliability Block Diagrams (DRBD)*

As the name implies, in this approach (Distefano & Puliafito 2007b; Distefano & Puliafito 2007a) reliability block diagrams are used to specify the redundancy structure of the system. The edges of the diagram are attributed with components, which are implicitly defined to have 3 states: active, standby and failed. Components in active and standby-state can fail (a transition to state failed occurs) whereas failed components can be repaired (modeled by a transition to the state active). Moreover, dependency edges can be defined by the modeler further specifying the behavior of the components and interdependencies between pairs of components. The DRBD-approach is very flexible: in total, 24 different kinds of dependencies between a pair of components can be defined. The approach also addresses the problem of conflicting dependencies, i.e. the behavior of the system in case of multiple dependencies influencing the same target at same time.

2.5 *Extended Fault Trees (eFT)*

Extended fault trees (Buchacker 1999; Buchacker 2000) allow for refining the leaves of traditional fault trees using finite automata. The modelers may either chose from a predefined set of automata (like components with several mutually exclusive failure modes) or create custom sub-models. Furthermore, it is possible to define interactions between different leaves of the fault tree. This is done by defining shared transitions in the respective automata. In such a way, events happening in one leave may affect the behavior of other leaves as well. Using predefined automata, inter-component dependencies can also be generated automatically. For instance, different types of failure propagation or varying failure rates are readily available.

2.6 *Boolean Driven Markov Processes (BDMP)*

In the approach Boolean logic driven Markov processes (BDMP, (Bouissou & Dutuit 2004; Bouissou & Maillard 2004)) each leave of the fault tree is refined using two Markov chains: one for normal operation and one for a so-called triggered operation. A trigger is a Boolean variable. Whenever a trigger changes from 'true' to 'false' or from 'false' to 'true', the affected component(s) change their state from one Markov chain to the other. Triggers are defined on the fault tree level as an edge between a source and a target gate. They are set to 'true' if the source gate is failed and set to 'false' otherwise. A trigger affects all components which are ancestors of the 'target'-gate. The BDMP-approach can be used to model a wide variety of inter-component dependencies, including cold, hot and warm standby, on demand failures, aging components, mutually exclusive failure modes, as well as common cause failures. It was shown that such an approach is more general than the DFT-approach (Bouissou 2007).

2.7 *OpenSESAME*

In our previous work we have developed the tool OpenSESAME (Simple but Extensive, Structured Availability Modeling Environment (Walter & Schneeweiss 2005; Walter et al. 2008; Walter 2000)).

In OpenSESAME, systems structures are specified using reliability block diagrams or fault trees. So called failure dependency diagrams (FDD) are used to describe failures with a common cause, failure propagation and so on. Furthermore, shared repair resources and non-zero fail-over times between redundant components can be modeled with OpenSESAME.

3 DEPENDABILITY MODEL SPECIFICATION

In this section we introduce our new formalism LARES for specifying dependability models. This formalism allows the user to define a wide variety of behaviors, starting from simple static reliability block diagrams or fault trees up to complex dynamic behavior including failure propagation, failures with common cause, prioritized repair strategies, etc..

3.1 Goals and principles

LARES has been devised with the following goals in mind:

- A dependability model should have a clear structure, consisting of modules with well-defined interfaces and realizing an encapsulation of internal information.
- The specification formalism should provide an easy-to-learn textual user interface and it should also provide an equivalent graphical user interface, generated with the help of state-of-the-art tools.
- The specification formalism should be designed having in mind that other kinds of system models or specifications will be automatically transformed to/from this formalism.

Our reliability model consists of one or more modules which are composed in a strictly hierarchical manner. There is always a single top-level system instance, representing the overall system. Usually, each module describes a part (component or subsystem) of the modeled system, but a module may also represent a different entity relevant to the modeled system, such as a repairman or some kind of external influence.

A module may have arbitrarily complex behavior. In the simplest non-trivial case, the behavior is Boolean, meaning that the module has two states, being either functional or failed. In the most general case, a module may have an underlying multi-dimensional state space, generated as the cross-product of several one-dimensional state spaces.

In general, the behavior of a module is composed of different aspects (also called behaviors or traits). As an example, one behavior could be the level of operation of the system (with states `fully_functional`, `degraded` and `failed`), and another aspect could be the way in which the module is used as a spare within the overall, reconfigurable system (with states `idle`, `used_by_subsys1`, `used_by_subsys2`). Each such behavior corresponds to one dimension of the module's state space, and a possible combined state would be the tuple (`degraded`, `used_by_subsys2`).

For defining the state space of a module, we exactly one so-called behavior description (`Behavior`) for each dimension of the state space. A behavior description can also be represented graphically by equivalent behavior diagrams, as exemplified below in the case studies. Behavior descriptions are finite state machines, whose transitions are labeled with a name and attributed with either delay information (in the case of timed transitions) or weight information (in the case of immediate, i.e. instantaneous transitions). The behavior diagram as a whole may carry parameters, such as delay parameters or branching probabilities.

Within a module, Boolean conditions, depending on the states of this module and of all its sub-modules, can be specified. These Boolean conditions, in turn, are used to guard transitions of the behavior descriptions, i.e. a transition may only take place if the corresponding guard condition is satisfied. Furthermore in each module, single instances or containers of instances of the modules types within the scope can be defined to express the internal structure of a module.

3.2 Specification language

In this section, the grammar definition is presented in a somehow "trimmed" version, just to point out the most important aspects.

The definition of a behavior has the following form:

```
Behavior <beh_name> {
  State state_1, state_2, ..
  Transitions from <state_name>
    if [<trans_cond>] -> <state_name>
         [, <delay_attrib> | <weight_attrib>]
  Transitions from ..
}
```

The name `beh_name` of a behavior will be referred to in the behavior list of those modules implementing this kind of behavior. Internally, the behavior description defines the set of states and all possible state transitions. It is also possible that a behavior inherits from another behavior.

A module type is described using the keyword `Module`, followed by the module's name. The list of the module's behaviors and possibly the name of a module with higher abstraction from which it inherits is separated by a colon.

```
Module <name> : [moduleref,]
               beh_1, beh_2, .. {
  <initial_settings>
  <sub_modules>
  <instance_definitions>
  <boolean_conditions>
  <guard_statements>
  <modification_statements>
  <measure_definitions>
}
```

The contents of a module will be explained next. Each module's weight and delay parameters can be initially set:

```
initially <param_name> = <arith_expr>
```

The internal structure of each module is represented by defining the contained module instances. Again, the keyword `initially` allows to declare the initial state for each behavior of the specified instances. For a single instance:

```
<instance_name> is instance of <module>
  initially <combined_state>
```

For defining a container of instances:

```
<container_name> consists of <n> instances {
  [<range>] of <module> initially <combined_state>
  ...
}
```

A Boolean condition is defined using the keyword `Condition`, followed by a Boolean expression:

```
Condition <cond_name> = <boolean_expression>
```

Such conditions are used to guard transitions of behavior descriptions.

```
<cond_name> guards <trans_cond>
```

A condition may also be employed to trigger the modification of weight or delay parameters:

```
<cond_name> causes
  [delay_attrib | weight_attrib] = <arith_expr>
```

Result measures of interest are specified using the keyword `Probability`.

```
Probability <measure_name> =
  [Steady_State(<state>) | Transient(<state>,t)]
```

The next section will show how these language elements can be employed for specifying dependability models with non-trivial behavior.

4 CASE STUDIES

4.1 *Phased mission system*

Phased mission systems are reconfigured not only after failures, but also according to predefined time intervals with deterministic or random duration. Their lifetime is thus divided into phases which cannot be modeled independently of each other, because e.g. a failure of a component in one phase might influence the behavior of the system in the following phases.

An example of a simple but non-trivial system is presented in (Bouissou & Dutuit 2004). There, it is modeled in two different ways: by a stochastic Petri net and by a Boolean Driven Markov Process (BDMP). In the following, we will present a model of the same system using LARES .

```
/******** Behavior Definitions ********/
Behavior B_NonRepairable {
    State Active, Failed
    Transitions from Active
      if [true]              → Failed, delay MTTF
}

Behavior B_Switch {
   State Closed, Open, SA_Open, SA_Closed
   Transitions from Open
     if [int_close]          → Closed, weight gamma
     if [int_close]          → SA_Open, weight 1−gamma
   Transitions from Closed
     if [int_open]           → Open, weight gamma
     if [int_open]           → SA_Closed, weight 1−gamma
     if [true]               → SA_Open, delay MTTF
}

Behavior B_Phases {
   State Phase1, Phase2, Mission_Acc, Failure
   Transitions from Phase1
     if [true]               → Phase2, delay tau1
     if [Fail_P1]            → Failure
   Transitions from Phase2
     if [true]               → Mission_Acc, delay tau2
     if [Fail_P2]            → Failure
}

/***** Overall System Definition *****/
System main : B_Phases {

   /********** Initialization *************/
   initially tau1 = exponential 100h
   initially tau2 = exponential 50h

   /******* Module−Type Definitions *******/
   Module M_Component: B_NonRepairable{
     initially MTTF = exponential 10000h
   }

   Module M_Switch: B_Switch{
     initially MTTF = exponential 10000h
     initially gamma = 0.05
   }

   /********* Submodule−Instances *********/
   s contains 5 instances {
     [1 .. 4] of M_Switch initially Closed
          [5] of M_Switch initially Open
   }

   a is instance of M_Component initially Active
   b is instance of M_Component initially Active


   /************* Conditions *************/
   ( (a.Failed | not s[2].Closed | not s[4].Closed) &
     (b.Failed | not s[1].Closed | not s[3].Closed) )
     | s[2].SA_Closed | s[3].SA_Closed
     guards Fail_P1

   a.Failed | b.Failed | not s[2].Closed |
     not s[3].Closed | not s[5].Closed
     guards Fail_P2

   a.Failed guards s[2].int_open
   b.Failed guards s[3].int_open
   Phase2   guards s[1].int_open, s[4].int_open, s
       [5].int_close

   Mission_Acc | Failure causes
     forall m in { a , b, s[1 .. 5] }
       { m.MTTF *= Infinity }

   /******** Measure Definitions ********/
   Probability MissionSuccess   = SteadyState(main.
       Mission_Acc)
   Probability Mission_Failure = 1−Mission_Success
   Probability Reliability      = Transient(main.
       Failure, t)
}
```

Listing 1. LARES model of the phased mission system.

Fig. 2 contains a schematic drawing of the system: it comprises two major components (labeled a and b) and five switches. During phase 1, switches `s[1]`,

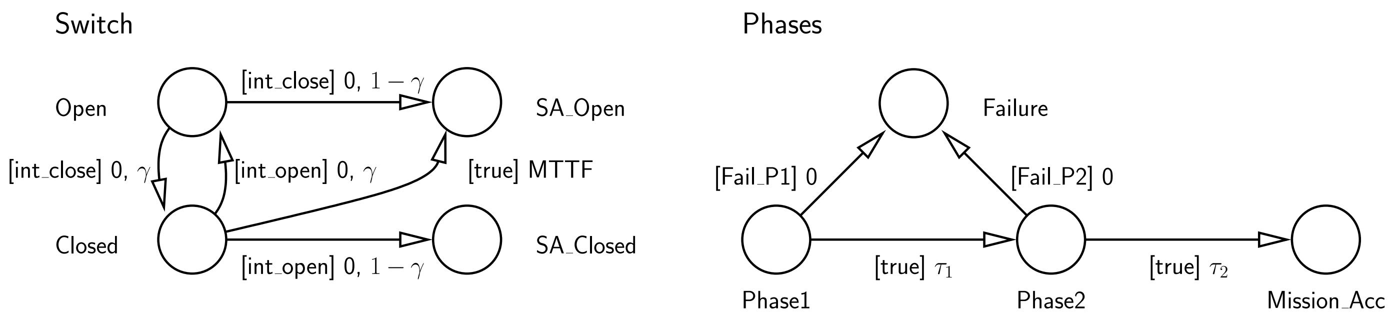

Figure 1. Behavior diagrams of the phased mission system.

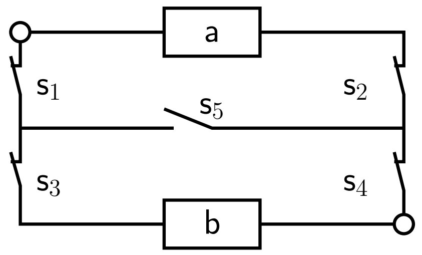

Figure 2. Phased mission system during phase 1 (no failures occurred).

s[2], s[3], and s[4] are all closed, whereas s[5] is in the open position. Components a and b are redundant and only one of them is needed for operation. However, a failure of either a and b is assumed to short-circuit the system which must be prevented by opening switch s[2] or s[3], respectively. An intentional opening of a switch fails with probability γ. Furthermore, unintentional opening of switches s[1..4] may also occur. Open switches (e.g. s[5] in phase 1) are assumed to be fault-free.

After an exponentially distributed time interval with mean $\tau_1 = 100$h, phase 1 ends and transition into phase 2 is made. In order to do so, first switches s[1] and s[4] are opened (unless they were opened unintentionally in phase 1), and then s[5] is closed. All three operations may fail with probability γ. During phase 2, a and b operate in series and a failure of either a,b, or an unintentional opening of s[2], s[5], or s[3] will lead to system failure. The mean length of phase 2 is $\tau_2 = 50$h, and it is exponentially distributed, too.

It is assumed that the system cannot be repaired and that all failure events occur with the same mean time to failure MTTF.

The textual LARES-model of the phased mission system is shown in Fig. 1. It consists of three behaviors, one module describing the overall system, and three measures.

The behavior NonRepairable defines a simple, non-repairable component which is used for the components a and b. The behavior Switch (see Fig. 1 (left), for a graphical representation) is used for the five switches. A switch can be intentionally opened or closed. Two guards (int_open and int_close) enable a state change from the closed to the open state and vice versa. However, with probability $1-\gamma$ a stuck-at fault occurs which is modeled using the states SA_Open and SA_Closed. Finally, in the closed position, the switch can fail by itself with mean time to failure MTTF.

The individual phases of the system are modeled using the behavior Phases, graphically represented in Fig. 1, right. The guards Fail_P1 and Fail_P2 describe situations which lead to system failure during phase 1 and phase 2, respectively.

In addition to these behaviors, the model comprises the component System which behaves according to Phases. It further defines that the system is made up of two non-repairable components a and b and five switches. Moreover, it describes the redundancy structure of the system during phase 1 and phase 2, and the interactions between the sub-modules. For example, it specifies that s[2] opens after a failure of component a, or that s[5] is closed at the beginning of phase 2. Finally, it sets the MTTF of all components to infinity (i.e. a failure can no longer occur), once the system is in the state Mission_Acc or in the failed state. The latter is done in order to decrease the size of the Markov chain defined by the model.

Finally, three measures Mission_Success, Mission_Failure and Reliability are defined.

4.2 *Multicomputer*

This is a slightly modified example taken from (Suñé & Carrasco 2001). It models a fault-tolerant multicomputer (see Fig. 3) comprising three computing modules (cm$_i$, $i \in \{1,2,3\}$). Each cm$_i$ comprises three CPU chips (cpuc), two port chips (ptc), and three memory modules (mm$_{i,j}$, $j \in \{1,2,3\}$). Each memory unit consists of ten memory chips (mc) and an interface chip (ic).

As the system is non-repairable, we do not involve transitions from failed to working states of a component. The failure rates of the CPU chips are identical for the CPUs on each computing module. However, each computing module is equipped with a different kind of chip. Therefore, the failure rate of the CPU chips depends on parameter i. The same holds for the port chips. On the contrary, the computing modules are identical in terms of memory. However, the memory modules within each computing module are dif-

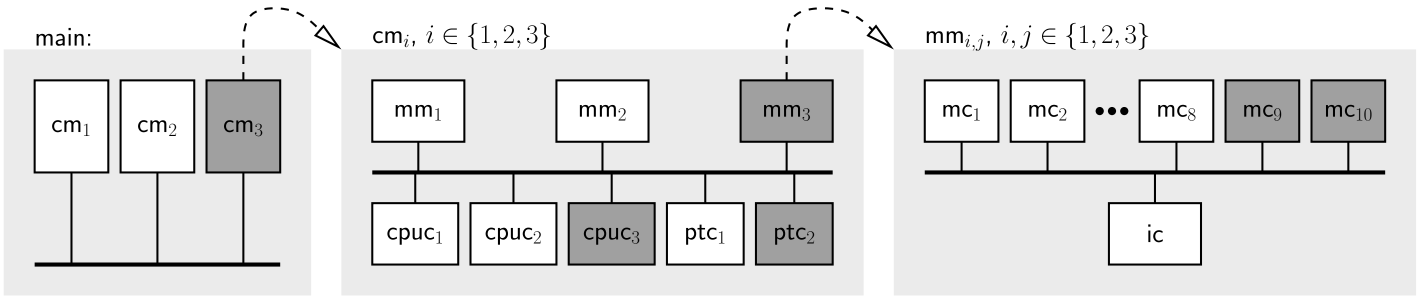

Figure 3. Hierarchical multicomputer. Grey blocks indicate spare-components with reduced failure rate during normal operation.

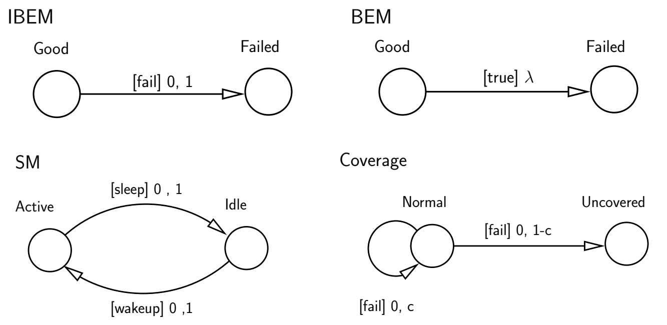

Figure 4. Behavior diagrams of the Multicomputer system.

ferent. Therefore, the failure rate of the memory and interface chips depends on parameter j.

According to (Suñé & Carrasco 2001), the system is working if at least two computing modules are working. In turn, a computing module requires two CPU chips, one port chip, and two memory modules for fault-free operation. Finally, a memory module is working if at least eight memory chips and its interface chip are fault-free. The undesirable event System_Failure can therefore be described using the following Boolean terms:

(1) `System_Failure` $\equiv$ `main.IBEM.Failed` $\leftarrow$ `2 oo cmi.IBEM.Failed`

(2) `cm.IBEM.Failed` $\leftarrow$ `2 oo mmi.IBEM.Failed | 2 oo cpuci.BEM.Failed | 2 oo ptci.BEM.Failed`

(3) `mm.IBEM.Failed` $\leftarrow$ `3 oo mci.BEM.Failed | ici.BEM.Failed`

Hereby, (1) corresponds to line 27 in Listing 2 and defines a system failure. (2) corresponds to line 46 and defines the failure of a computing module. Finally, (3) corresponds to line 90 and defines a memory module failure.

The dynamic behavior of the system is, again following (Suñé & Carrasco 2001), defined as follows:

1. The system parts shown in grey in Fig. 3, i.e. cm_3, $\text{cpuc}_{i,3}$, $\text{ptc}_{i,2}$, $\text{mc}_{i,j,9}$, and $\text{mc}_{i,j,10}$ are spare components which are only activated if they are needed for system operation. While not needed, the failure rate of all components belonging to the respective part of the system is reduced.
2. In all cases, recovery may fail itself, leading to an immediate failure of the corresponding (sub-)system. In detail, a failure of one of the ten $\text{mc}_{i,j,*}$ is not covered with probability $1 - c_{mc}$, leading to a failure of $\text{mm}_{i,j}$. A failure of one of the $\text{mm}_{i,*}$, $\text{cpuc}_{i,*}$, or $\text{ptc}_{i,*}$ is not recovered with probability $1 - c_{mm}$, $1 - c_{cpuc}$ or $1 - c_{ptc}$, respectively, and leads to a failure of cm_i. Finally, if one of cm_* fails, the overall system fails immediately with probability $1 - c_{cm}$. Note that also the failure of unused spare components can affect the system in such a way.

The failure behavior of a (sub-)system is modeled by using the "boolean error model" (BEM) for failures after an exponentially distributed time as well as the "immediate BEM" (IBEM) for triggered failure transitions, which is needed to push a failure in subsystems bottom-up (see Fig. 4, for the shortened textual specification see Listing 2).

In order to model issue 1, we use the spare model behavior SM. A subsystem which has this behavior can be `Active` or `Idle`, meaning that the subsystem is used for the overall operation or is functioning as spare. For instance, computing module cm_3 is initially spare, which means that all of its submodules are idle. If cm_3 is activated after failure of cm_1 or cm_2 then all of its submodules are activated recursively, except those which are still designated to work as spare component. For example the modules $\text{cpuc}_{3,1}$, $\text{cpuc}_{3,2}$ and $\text{ptc}_{3,1}$ are activated and their failure rate is increased immediately. The memory modules $\text{mm}_{3,1}$ and $\text{mm}_{3,2}$ are activated in such a way that all of their submodules, which are necessary for the correct functioning of these modules are activated. On the other hand the modules $\text{cpuc}_{3,3}$, $\text{ptc}_{3,2}$ and $\text{mm}_{3,3}$ remain idle, if all the other activated submodules of module cm_3 are not yet failed.

```
Behavior IBEM {
  State Good, Failed
  Transitions from Good
     if [fail] → Failed
}
Behavior Coverage {
  State Normal, Uncovered
  Transitions from Normal
     if [fail] → Normal, weight c
     if [fail] → Uncovered, weight 1−c
}
Behavior SM {
  State Active, Idle
  Transitions from Active
     if [sleep]  → Idle
  Transitions from Idle
     if [wakeup] → Active
}

System main : IBEM, Coverage {
  initially Coverage.c = 0.9

  /*********** Instances ************/
  cmi contains 3 instances of cm

  /******* System Failure **********/
  2 oo cmi.IBEM.Failed guards IBEM.fail

  /**** Coverage Reconfiguration ****/
  1 oo cmi.IBEM.Failed guards Coverage.fail , cmi
       [3].SM.wakeup
  Coverage.Uncovered guards IBEM.fail

   Behavior Cov_cpuc : Coverage {}
   Behavior Cov_ptc : Coverage {}
   Behavior Cov_mm : Coverage {}

    Module cm : IBEM, SM, Cov_cpuc, Cov_ptc,
         Cov_mm {
      initially Cov_cpuc.c = 0.95, Cov_ptc.c =
           0.98, Cov_mm.c = 0.99

      /*********** Instances ************/
      mmi contains 3 instances of mm
      cpuci contains 3 instances of cpuc
      ptci contains 3 instances of ptc

      /******* Module Failure **********/
      2 oo mmi.IBEM.Failed | 2 oo cpuci.BEM.Failed
            | 2 oo ptci.BEM.Failed guards IBEM.fail


      /**** Coverage Reconfiguration ****/
      1 oo cpuci[1 .. 2].BEM.Failed & SM.Active
           guards Cov_cpuc.fail , cpuci[3].SM.wakeup
      ptci[1].BEM.Failed & SM.Active guards
           Cov_ptc.fail , ptc[2].SM.wakeup
      1 oo mmi[1 .. 2].IBEM.Failed & SM.Active
           guards Cov_mm.fail , mm[3].SM.wakeup
      Cov_cpuc.Uncovered | Cov_ptc.Uncovered |
           Cov_mm.Uncovered | guards IBEM.fail

      /*** SpareModel Reconfiguration ****/
     SM.Idle guards mmi[1 .. 3].SM.sleep , cpuci[1
           .. 3].SM.sleep , ptci[1 .. 3].SM.sleep
     SM.Active & ptci[1].BEM.Good guards ptci[1].
           SM.wakeup
     SM.Active & 2 oo mmi[1 .. 2].IBEM.Good
           guards mmi[1 .. 2].SM.wakeup
     SM.Active & 2 oo cpuci[1 .. 2].BEM.Good
           guards cpuci[1 .. 2].SM.wakeup

     Behavior BEM {
        State Good, Failed
        Transitions from Good
           if [true] → Failed , delay exponential
                lambda
     }
      Module cpuc : BEM, SM {}
      Module cpuc1 : cpuc {
        SM.Active causes BEM.lambda = 6
        SM.Idle causes BEM.lambda = 3
      }
      Module cpuc2 : cpuc {...}
      Module cpuc3 : cpuc {
        SM.Active causes BEM.lambda = 4
        SM.Idle causes BEM.lambda = 2
      }
      Module ptc : BEM, SM {}
      Module ptc1 : ptc {
        SM.Active causes BEM.lambda = 4
        SM.Idle causes BEM.lambda = 2
      }
      Module ptc2 : ptc {...}
      Module ptc3 : ptc {...}
      Module mm : IBEM, SM, Coverage {
        initially Coverage.c = 0.8

        /*********** Instances ************/
        mci contains 10 instances of mc
        ici is instance of ic initially BEM.Good, SM
             .Active

        /******* Module Failure **********/
        3 oo mci.BEM.Failed | ici.BEM.Failed
             guards IBEM.fail

        /**** Coverage Reconfiguration ****/
        Condition one_fail = 1 oo mci[1 .. 8].BEM.
             Failed & 7 oo mci[1 .. 8].BEM.Good &
            SM.Active
        one_fail & mc[9].BEM.Good guards Coverage.
             fail , mc[9].SM.wakeup
        one_fail & mc[9].BEM.Failed guards
             Coverage.fail , mc[9 .. 10].SM.wakeup
        2 oo mci[1 .. 8].BEM.Failed & SM.Active
             guards Coverage.fail , mc[9 .. 10].SM.
             wakeup
        Coverage.Uncovered guards IBEM.fail

        /*** SpareModel Reconfiguration ****/
        SM.Idle guards mci[1 .. 10].SM.sleep
        SM.Active & 8 oo mci[1 .. 8].BEM.Good
             guards mci[1 .. 8].SM.wakeup

         Module ic : BEM, SM {
           SM.Active causes BEM.lambda=8
           SM.Active causes BEM.lambda=4
         }
         Module mc : BEM, SM {}
         Module mc1 : mc {
           SM.Active causes BEM.lambda = 6
           SM.Idle causes BEM.lambda = 3
         }
         Module mc2 : mc {...}
         Module mc3 : mc {...}
      }

      Module mm1 : mm {
        mci[1 .. 10] are instances of mc1  {
           [1 .. 8]  initially BEM.Good, SM.Active
           [9 .. 10] initially BEM.Good, SM.Idle
        }
      }
      Module mm2 : mm {
        mci[1 .. 10] are instances of mc2 {...}
      }
      Module mm3 : mm {...}
    }

    Module cm1 : cm {
      cpuci[1 .. 3] are instances of cpuc1 {
           [1 .. 2] initially BEM.Good, SM.Active
           [3]      initially BEM.Good, SM.Idle
      }
      mmi[1 .. 3] are instances of mm1 {
         [1 .. 2] initially BEM.Good, SM.Active
         [3]      initially BEM.Good, SM.Idle
      }
      ptci[1 .. 2] are instances of ptc1 {
          [1] initially BEM.Good, SM.Active
          [2] initially BEM.Good, SM.Idle
      }
    }
    Module cm2 : cm {...}
    Module cm3 : cm {...}

    cmi[1] is instance of cm1 initially
               IBEM.Good, SM.Active, Coverage.
                   Normal
    cmi[2] is instance of cm2 initially
               IBEM.Good, SM.Active, Coverage.
                   Normal
    cmi[3] is instance of cm3 initially
               IBEM.Good, SM.Idle, Coverage.Normal
}
```

Listing 2. LARES model of the multicomputer system.

Dynamic behavior 2 is modeled by using the `Coverage` behavior. For instance if one of the active cm$_i$ fails then this failure could remain undetected leading to the state `Uncovered` within the main modules `Coverage` behavior. That, in consequence, results in a system failure (see line 31 in Listing 2). The module `cm` has three different `Coverage` behaviors in order to parametrize the (possibly different) probabilities c_{cpuc}, c_{ptc} and c_{mm} for detecting a failure of one of its submodules `cpuc`, `ptc` resp. `mm`. For that purpose we use the inheritance property of Behaviors (see lines 33 - 38).
Specialized subtypes of abstract module types can be specified as illustrated in lines 65 - 74. There, `cpuc_1`, `cpuc_2` and `cpuc_3` inherit all Behaviors, Guards- and Cause-Conditions of the abstract module `cpuc` and are specialized by the newly defined conditions. The concrete instances of these abstract modules are defined as `cpuci` in the modules `cm_1`, `cm_2` and `cm_3` (see lines 128 - 143). In this way it is possible to build up a hierarchy of modules with different (sub-)module types in each hierarchy layer.

5 CONCLUSION & OUTLOOK

In this paper we have introduced the LARES formalism for the specification of dependability models of reconfigurable systems. The formalism follows a structured hierarchical approach and includes features that allow the user to describe arbitrarily complex dynamic behavior. The paper described the textual user interface, concretized by two examples from the literature. Next steps will include the definition of a formal semantics for LARES. Based on this semantics, we will develop model transformations in order to automatically convert different application-specific modeling formalisms to the LARES formalism, and implement interfaces to existing analysis engines, which will be used to perform the quantitative analysis of the specified models.

REFERENCES

Blum, A., Goyal, A., Heidelberger, P., Lavenberg, S., Nakayama, M., & Shahbuddin, P. (1994). Modeling and analysis of system dependability using the system availability estimator. In *FTCS 1994*, pp. 137–141.

Blum, A., Heidelberger, P., Lavenberg, S. S., Nakayama, M. K., & Shahabuddin, P. (1993). *"System Availability Estimator (SAVE) Language Reference and User's Manual Version 4.0"*. IBM Research Report RA 219S.

Boudali, H., Crouzen, P., Haverkort, B. R., Kuntz, M., & Stoelinga, M. (2008). Architectural dependability evaluation with arcade. In *DSN 2008*, pp. 512–521.

Boudali, H., Crouzen, P., & Stoelinga, M. (2007a). A Compositional Semantics for Dynamic Fault Tree Analysis in terms of Interactive Markov Chains. In *ATVA'07*, LNCS, pp. 708–717. Springer.

Boudali, H., Crouzen, P., & Stoelinga, M. (2007b). Dynamic Fault Tree Analysis Using Input/Output Interactive Markov Chains. In *DSN 2007*, pp. 708–717. IEEE Computer Society.

Bouissou, M. (2007). A generalization of Dynamic Fault Trees through Boolean logic Driven Markov Processes (BDMP). In *ESREL 2007*, Volume 2, pp. 1051–1058. Taylor & Francis Ltd.

Bouissou, M. & Dutuit, Y. (2004). Reliability analysis of a dynamic phased mission system. In *MMR 04*.

Bouissou, M. & Maillard, S. (2004). Set of test-cases in the field of system dependability assessment.

Buchacker, K. (1999). Analyzing Safety Critical Systems Using Extended Fault Trees. In *ARCS 1999*, pp. 117–125. Gesellschaft für Informatik.

Buchacker, K. (2000). Modelling with extended fault trees. In *HASE 2000*, pp. 238–246. IEEE.

Coppit, D. & Sullivan, K. (2000). Galileo: A tool built from Mass-Market Applications. In *ICSE '00*, pp. 750–753. IEEE Computer Society Press.

Distefano, S. & Puliafito, A. (2007a). Dependability modeling and analysis in dynamic systems. In *IPDPS '07*.

Distefano, S. & Puliafito, A. (2007b). Dynamic reliability block diagrams: Overview of a methodology. In *ESREL '07*, pp. 1059–1068.

Dugan, J., Bavuso, S., & Boyd, M. (1992). Dynamic fault-tree models for fault-tolerant computer systems. *IEEE Trans. on Rel. 41*(3), 363ff.

Dugan, J., Sullivan, K., & Coppit, D. (2000). Developing a low-cost high-quality software tool for dynamic fault-tree analysis. *IEEE Trans. on Rel. 49–59*(1), 49ff.

Dugan, J., Venkataraman, B., & Gulati, R. (1997). DIFTree: A software package for the analysis of dynamic fault tree models. In *RAMS '97*, pp. 64–70.

Goyal, A., Carter, W., de Souza e Silva, E., Lavenberg, S., & Trivedi, K. S. (1986). The System AVailability Estimator (SAVE). In *FTCS 1986*. IEEE Computer Society Press.

Manian, R., Dugan, J., Coppit, D., & Sullivan, K. (1998). Combining various solution techniques for dynamic fault tree analysis of computer systems. In *HASE '98*, pp. 21–28.

Suñé, V. & Carrasco, J. (2001). A failure-distance based method to bound the reliability of non-repairable fault-tolerant systems without the knowledge of minimal cuts. *IEEE Trans. on Rel. 50*(1), 60–74.

Walter, M. (2000). OpenSESAME - A Tool's Concept. In *ICALP '00*, Volume 8 of *Proceedings in Informatics*, pp. 477–484. Carleton Scientific.

Walter, M. & Schneeweiss, W. (2005). *The modeling world of Reliability/Safety Engineering*. LiLoLe Verlag.

Walter, M., Siegle, M., & Bode, A. (2008). OpenSESAME - The simple but extensive, structured availability modeling environment. *Reliability Engineering and System Safety 93*(6), 857–873.