A LAnguage for REconfigurable dependable Systems: Semantics & Dependability Model Transformation

Martin Riedl
Institut für Technische Informatik,
Universität der Bundeswehr München,
Werner-Heisenberg-Weg 39, 85579 Neubiberg, Germany
martin.riedl@unibw.de

Markus Siegle
Institut für Technische Informatik,
Universität der Bundeswehr München,
Werner-Heisenberg-Weg 39, 85579 Neubiberg, Germany
markus.siegle@unibw.de

$LARES$ (LAnguage for REconfigurable dependable Systems) has been defined to model fault-tolerant systems. It serves as an easy-to-learn formalism that allows to describe the structure of a system and to express its dynamic failure and repair behavior in a convenient and concise way. The paper gives insight into the recent improvements to the language and its formal transformation semantics into a stochastic process algebra.

1. INTRODUCTION

Fault-tolerant systems are most often designed by developers and engineers from industry. Since those systems have to satisfy some important non-functional properties (e.g. reliability, availability, survivability), tools have to be applied to specify and consequently analyze those systems. Exhaustive methods or rare-event simulations are the most suitable methods to be applied, as cases of failures are "hopefully" intrinsically extremely rare. Tools which implement exhaustive methods mostly come from academia. There, research is mainly focused on expressiveness and theoretic verifiability of models specified with the help of formal languages, or speeding up the analysis methods. Indeed, formalisms such as reliability block diagrams (RBD) or fault trees are used by engineers to design fault-tolerant systems but suffer from limited expressiveness. State-based formalisms such as Petri nets or Markov models do not pervade common usage. Compositional languages such as process algebra seem to fill this gap but still do not attract sufficiently - probably because it is a non-trivial task to specify process synchronizations correctly.

A language that provides both, i.e. basic well-known concepts to model simple fault-tolerant aspects and at the same time maintaining expressive power to specify more complex dependencies, has been introduced as $LARES$ in [Gouberman et al. (2009)]. A number of improvements and advanced features have been added since then. Models can be specified in a hierarchical way by so-called `Module` or `Behavior` definitions. Instances inside module definitions can be defined by either inheritance or applying an `Instance` statement. `Condition` statements denote Boolean expressions similar to RBDs and are used to define the interaction between those instances by `guards` statements. The direction of information flow within a model has to be defined stringently, using either `Condition` or `forward` definitions. Equally, scoping and visibility restrictions are provided to assure modularity and clarity.

The aim of this paper is to provide information on the $LARES$ language expressiveness and semantics (Sec. 3) and the fully automated transformation of $LARES$ dependability models into the CASPA process algebra [Kuntz, Siegle and Werner (2004); Bachmann et al. (2009)] (Sec. 4) in order to allow formal analysis (Sec. 5). Thus, the contribution of the paper consists of the presentation of a complete modelling framework, starting from an expressive and user-friendly specification language and resulting in a formal quantitative model and its automated analysis.

2. RELATED WORK

The ideas leading to the $LARES$ transformation semantics have been collected in [Riedl et al. (2010)] and successive works. There, ZuVerSicht models, providing a basic non-Boolean error model with aspects such as error propagation, common cause failure and two kinds of repair concepts, were translated into a stochastic process algebra and analyzed by the CASPA tool.

After $LARES$ had been defined initially [Gouberman et al. (2009)], further work to emphasize the suitability of modeling fault-tolerant systems using a dialect of $LARES$ has been described in [Walter and Lê (2011); Walter (2011b,a)]. The objective of $LARES$ is twofold, to serve as an intermediate language as well as a user-level language. In [Lê and Walter (2011)] an efficient conversion from conventional graph based reliability block diagram structures to $LARES$ Boolean expressions is described and will contribute to the $LARES$ framework within a graphical editor which is currently under development (see Sec. 6).

Related work has been done in the COMPASS project [Bozzano et. al. (2011)] by formalizing a subset of AADL[1] incorporating also the error-model annex. There, the semantics is given in terms of (Networks of) Event-Data Automata. Arcade [Boudali et. al. (2008)] has taken a similar approach by formalizing a subset of AADL using IO-IMCs (Input/Output Interactive Markov Chains). Automated model transformation using the *model driven development* paradigm has been applied in [Grassi et al. (2008)] where an intermediate language called KLAPER (Kernel LAnguage for PErformance and Reliability analysis) was defined.

3. LARES EXPLAINED BY EXAMPLE

$LARES$ serves as a generic language in which any kind of discrete-event stochastic system can be specified. However, modellers benefit most if they use $LARES$ for describing systems which have a hierarchic structure, where events depend on combinations of causes and may lead to combinations of consequences.

As a running example, a fault-tolerant network system is given. It consists of two processors and a network that connects those processors using a number of redundant links (see also Fig. 1). Each of the processors can fail as well as each of the links of the network. Initially only the first link is active and the others remain inactive. However, if the active link fails, the second

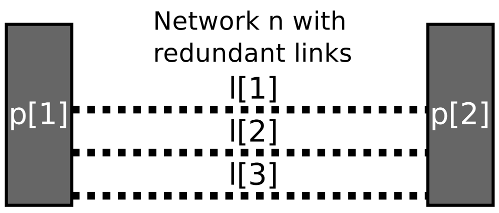

Figure 1: *Redundant Network Example*

is activated and so forth. Once all links are failed, the network as a whole is failed. The system itself fails if one of the processors or the network fails. In the following, explanations are given how this model can be specified using LARES. We start with the general system structure as shown in Listing 1. In $LARES$ a keyword `System` is provided to denote the system instance of the fault-tolerant network model by a name, i.e. `FTN`. Since our interest is on the system level, i.e. whether the system as a whole is operable, we provide a failure behavior `BSys` to the system definition. Inside the system definition we provide all instantiations of its subcomponents, i.e. two processors `p[1]` and `p[2]` by stating `Instance p[i] of Processor` with an expansion expression of this statement for the variable `i` ranging between 1 and 2 and the network instance n by stating `Instance n of Network(numLink=3)`, meaning that we provide the value 3 for the parameter `numLink` of the Network definition.

```
Behavior BSys {...}

Behavior BLink {...}

Behavior BProcessor {...}

System FTN : BSys {
  Module Processor : BProcessor {...}
  ...
  expand (i in {1 .. 2}) {
    Instance p[i] of Processor
  }
  Instance n of Network(numLink=3)
}

Module Network(numLink) {
  Module Link : BLink {...}
  Instance l[1] initially active of Link
  expand (i in {2 .. numLink}) {
    Instance l[i] initially passive of Link
    ...
  }
}
```

Listing 1: Fault-Tolerant Network System: Structure

The processor definition itself has a behavior which is defined on the root level of the model. Inside the network definition also the subinstances of the Links are created. There, the statement `Instance l[1] initially active of Link` creates the active link instance, while the remaining links depending on the number of links of the corresponding variable `numLink` are created being passive initially. The link definition is also stated in the network definition, while its corresponding behavior is stated on the root level of the model. Another important aspect concerns the visibility of definitions. In $LARES$ the scope of behavior and module definitions behaves as in object-oriented programming with (inner-)class definitions, i.e. a behavior/module definition is visible at the same level where it is defined and at the subtrees. Thinking of our example as shown in Fig. 2: the definition `Link` is stated inside a definition `Network` and therefore not visible inside another definition, e.g. `FTN` and the subtree belonging to `FTN`. That means, a link instance cannot be created within the `FTN` definition or in its subtree. Note, that the self-referring dashed arrow denotes that system module definition `FTN` has a twofold meaning, i.e. it serves as both, a definition and its corresponding instantiation. So far, the structure of the model is

[1]Architecture Analysis and Design Language (AADL): `http://www.aadl.info`

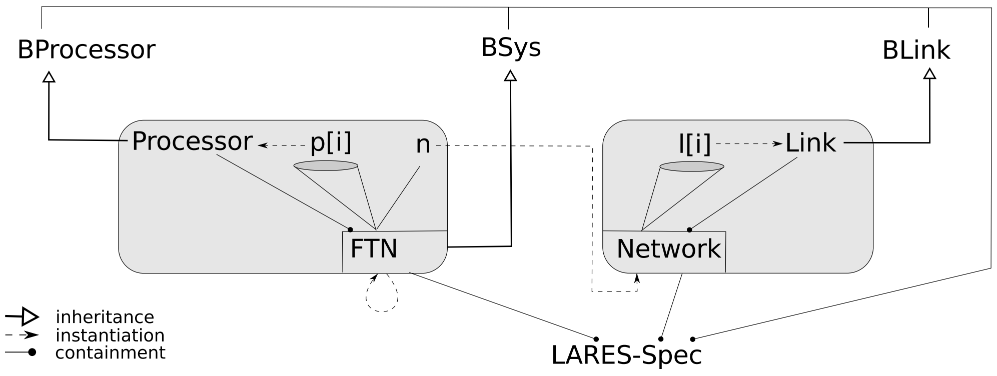

Figure 2: *Structural Model: Scoping*

defined, i.e. all module/behavior definitions are given including their instances which implicitly define the resulting instance tree. Next the interactions of the instances have to be complemented. For that purpose, a number of statements can be defined within a module definition:

- `Conditions`, i.e. boolean terms as assertions over states or other conditions.
- `forward` rules (which propagate an event and define which kind of synchronised behavior or which remote `forwards` should be triggered by referring to their label).
- `guards` statements (which generate such an event and define which kind of synchronized behavior or which `forwards` should be triggered).

The general flow of information is depicted in Fig. 3: the information over assertions on states is aggregated using a `Condition` statement, which generates an event within the `guards` statement. This in turn is forwarded by `forward` statements to guarded transitions (in case of a number of intermediate levels), or triggering a behavior directly by referring to the guard label of the guarded transition.

In Listing 2 a $LARES$ specification of the example is provided. One can approach the completion of the model from different directions (either a top-down, or bottom up, or mixed approaches). Here a top-down approach has been chosen, which means that we start from the system definition, where we state how its subtrees should interact.

3.1. System Module Definition

The initial state of the system behavior is defined to be in `ok`, i.e. the statement `Initial ok initially BSys.sok` defines that the behavior instance of the system has to be in `sok` initially. Remember that we have the following redundancy structure of the system: the system fails if one of the processors fail, or the network fails. Therefore the statement `OR[i in 1 .. 2] p[i].failed | n.nfailed guards BSys.<systemfail>` is formulated. It guards a label referring to a transition of the systems behavior instance `BSys`. Only two states `sok` and `sfailed` are explicitly defined and its corresponding transitions: a guarded transition that is triggered by the systemfail event and a transition that results in the implicitly defined state `shazard` after sojourning in `sfailed` by an exponentially distributed delay.[2]

3.2. Network Module Definition

In the network we have to additionally state that if a link is failed, the next one has to take over. This is expressed by the statement `l[i-1].lfailed guards l[i].<swactive>`. The expand expression implies that this guard statement is duplicated depending on the `numLink` parameter, i.e. since `numLink` is 3 the expand results in two guard statements meaning that that if `l[1]` is failed `l[2]` is signalled to switch active (`<swactive>`) and that if `l[2]` is failed, `l[3]` has to switch active. In addition to define how the links interact due to failures, we have to state whether the network has failed and make its internal state visible to the environment. This is done using a `Condition` statement. `Condition nfailed = AND[i in 1 .. numLink] l[i].failed` states that the network is failed (i.e. labelled by `nfailed`) if all links are in their failed state. Due to the fact that there is no inherited behavior or an incompletely initialized subinstance no explicit `Initial` statement has to be provided.

3.3. Link Module/Behavior Definition

The link module definition consists of a few simple statements, the `forward` statement to pass the signal `<swactive>` to the link's behavior. Besides, offering the two initial states `active`, i.e. if `BLink` is in state

[2]Apart from transitions with timed delay, LARES also offers weighted immediate transitions which may lead to vanishing states similar to GSPNs.

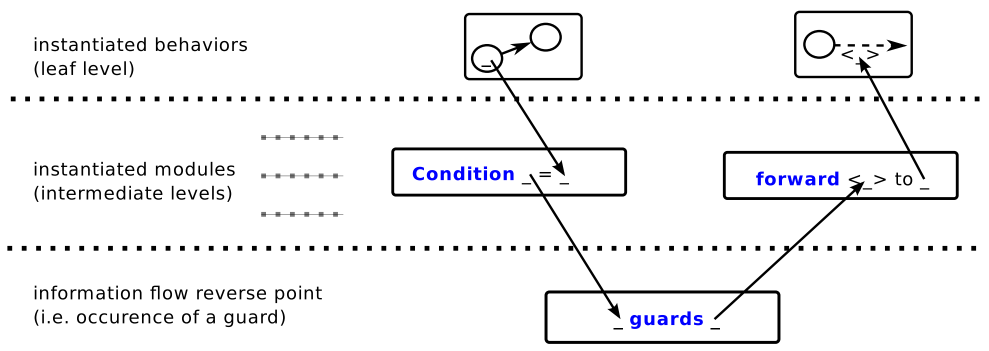

Figure 3: *General Flow of Information*

`lactive`, and `passive` i.e. if `BLink` is in state `lok`. The corresponding behavior to a link `BLink` is defined using three states `lactive`, `lok` and `lfailed` with the corresponding transitions in between them: a guarded transition from `lok` to `lactive` with the guard label `<swactive>`. And the unguarded transitions from either `lok` or `lactive` to `lfailed` with distinct exponentially distributed rates.

3.4. Processor Module/Behavior Definition

After all interesting parts of the model have been explained, for the sake of completeness, the behavior and the module definition of a processor remain to be provided. In the module definition of the processor a `Condition pfailed` states that its behavior instance `BProcessor` is in state `pfailed`. With `Initial ok initially BProcessor.pok` an initial state is defined for a processor, i.e. sets the initial state of the behavior instance to `pok`. By mentioning these two states, all states of the Behavior `BProcessor` have been considered. In addition, an unguarded transition with an exponential distributed delay can move the behaviors state from `pok` to `pfailed`.

```
Behavior BSys {
    State sok, sfailed
     Transitions from sok if <systemfail> → sfailed
     Transitions from sfailed
      if <true> → shazard , delay exponential 2
}

Behavior BLink {
    Transitions from lok
      if <swactive> → lactive
      if <true> → lfailed , delay exponential 0.1
    Transitions from lactive
      if <true> → lfailed , delay exponential 0.3
}

Behavior BProcessor {
     Transitions from pok
      if <true> → pfailed , delay exponential 0.005
}

System FTN : BSys {
  Module Processor : BProcessor {
      Condition pfailed = BProcessor.pfailed
      Initial ok = BProcessor.pok
  }
  expand (i in {1 .. 2}) {
     Instance p[i] of Processor
  }
  Instance n of Network(numLink=3)
  Initial ok = BSys.sok
 OR[i in {1 .. 2}] p[i].pfailed |
     n.nfailed guards BSys.<systemfail>
}

Module Network(numLink) {
   Module Link : BLink {
     forward <swactive> to BLink.<swactive>
     Condition lfailed = BLink.lfailed
     Condition passive = BLink.lok
     Initial active = BLink.lactive
     Initial passive = BLink.lok
  }
  Instance l[1] initially active of Link
  expand (i in {2 .. numLink}) {
     Instance l[i] initially passive of Link
    AND[j in {1 .. i-1}] l[j].lfailed &
       l[i].passive guards l[i].<swactive>
  }
  Condition nfailed =
    AND[i in {1 .. numLink}] l[i].lfailed
}
```

Listing 2: Fault-Tolerant Network System: Full Specification

In the example *conditional forwards* have not been addressed. With their help an additional boolean expression can be formulated pertaining to the state of its subtree. The `forward` statement in consequence only affects the behavior of its subtree if the condition is fulfilled. Another important aspect to note is, that both the guard statement and the `forward` statement follow a **max-sync** semantics: all addressed instantiated subbehaviors or submodules that are able to follow (as soon as a certain condition triggers a related guard statement at the upper level) will do this instantaneously in a synchronous step.

4. TRANSFORMATION SEMANTICS

This section is split into two parts. The first part considers in-model transformations, i.e. the resolution of certain language features that lead to an expanded but behavior-equivalent $LARES$ model called $LARES_{BASE}$. The second part of the section is specific to the transformation into the destination formalism. In this paper the focus is on the transformation into the CASPA stochastic process algebra language [Bachmann et al. (2009)]. Other target formalisms have also been addressed (see Sec. 6).

4.1. In-model Transformation

Three consecutive steps lead to a $LARES_{BASE}$ model: At first all instantiation parameters have to be resolved, leading to different realizations of behavior or module definitions depending on their instantiation parameterizations (see Fig. 4). Secondly, within Boolean expressions used in `forward`, `guards` and `Condition` statements, references to other conditions can be stated. All of those references are resolved such that a Boolean expression only consists of references to states. And thirdly, all guard label forwards are resolved, such that only modified `guards` statements remain with direct references to the guard labels inside the instantiated behavior definitions. In the following paragraphs, the steps are explained in detail by means of the running example.

4.1.1. Parameter Expansion

Starting at the root node of a $LARES$ model we first identify all parameters defined in the system definition `FTN`. We then search for all `expand` statements. The one we can find in the system instance of the running example is:

```
expand (i in {1 .. 2}) {
   Instance p[i] of Processor
}
```

Since the set expression used in the statement is static (i.e. does not depend on a parameter in this case), the expansion can take place immediately, resulting in:

```
Instance p[1] of Processor
Instance p[2] of Processor
```

The same holds for the `guards` statement within the system instance which is expanded to:

```
(
   p[1].failed | p[2].failed
) | n.nfailed guards BSys.<systemfail>
```

Since there is nothing more to expand, a recursive descent is performed over the subinstances. In consequence, the scheme is applied to the module definition that is referred to by the instantiation with the given parameters. Of course, the definition has to be in scope (i.e. must not be defined in another subtree of the definitions). Note that the network instance is created by setting the `numLink` parameter to 3. Assume that we perform now the parameter expansion on the network instance. The body of the module's definition contains an `expand` statement over a statement block. Since the instance of this definition has a defined parameter set, the set expression can be evaluated and the statement block expanded corresponding to the evaluation. The statements in the blocks are also evaluated concerning the iterators, resulting in:

```
Instance l[2] initially passive of Link

(
   l[1].lfailed & l[2].passive
) guards l[2].<swactive>

Instance l[3] initially passive of Link

(
   ( l[1].lfailed & l[2].lfailed ) & l[3].passive
) guards l[3].<swactive>
```

What remains is the `Condition` that states whether the network is failed. This condition is expanded to:

```
Condition nfailed = (
  l[1].failed & l[2].failed & l[3].failed
)
```

The same schema is performed also on the link and the processor instances. Since there are no parameters defined, no specific expansions have to be performed. However, it is important to note that all expanded definitions contain concrete parameters and are directly associated with their instantiations. For the sake of brevity, it is not detailed how the initials are propagated through the model, but it follows the same scheme as the propagation of the parameter evaluations.

4.1.2. Condition Expansion

Since the parameter expansion provides us with a concrete instance tree (see Fig. 4) and its associated expanded definitions, all Boolean expressions can be resolved recursively. Starting at the leaves of the instance tree, all condition statements found in the expanded definitions are rebuilt in such a way that there is no dependency to other conditions any more. As an example given:

```
Condition bothGood = b1.Good & b2.Good

Condition allGood = bothGood | b3.Good
```

results in

```
Condition bothGood = b1.Good & b2.Good

Condition allGood = (b1.Good & b2.Good) | b3.Good
```

Of course things are a bit more complicated since resolving local dependencies is sensitive to the order of resolvable conditions. Therefore, such resolutions have to follow a *topological order* concerning their dependencies, similar to the evaluations of set expressions within the `expand` statements. This aspect won't be detailed since this would go beyond the scope of the paper. Since a `Condition` of a module definition of an instance is visible to the environment, as a result of the recursive descent, all modified conditions are provided. Going back to the example, this means that for the network `n` the corresponding condition definitions

```
(
   l[1].lfailed & l[2].passive
) guards l[2].<swactive>

(
   (l[1].lfailed & l[2].lfailed) &
   l[3].passive
) guards l[3].<swactive>
```

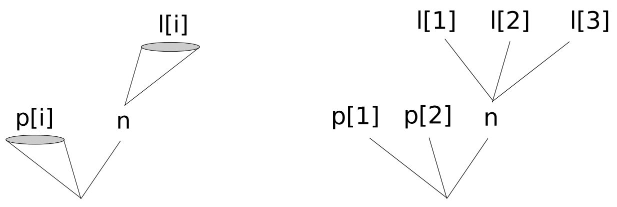

Figure 4: *Parameter Expansion: From an Abstract to a Concrete Instance Tree*

```
Condition  nfailed = (
  l[1].failed & l[2].failed & l[3].failed
)
```

refer to the condition `lfailed` of different instances of the links. Taking the results of the recursive descent, the conditions of the network will then be transformed into new Boolean expressions that only consist of references to states of instantiated behaviors:

```
(
  l[1].BLink.lfailed & l[2].BLink.lok
) guards l[2].<swactive>

(
  (l[1].BLink.lfailed & l[2].BLink.lfailed) &
  l[3].BLink.lok
) guards l[3].<swactive>

Condition  nfailed = ( l[1].BLink.lfailed &
  l[2].BLink.lfailed & l[3].BLink.lfailed
)
```

As a result, also the Boolean expressions used at the level of the system instance are modified

```
(
  (
    p[1].BProcessor.pfailed |
    p[2].BProcessor.pfailed
  ) | ( n.l[1].BLink.lfailed &
    n.l[2].BLink.lfailed & n.l[3].BLink.lfailed
  )
) guards BSys.<systemfail>.
```

4.1.3. Guard Expansion

The structure of a `guards` statement is similar to that of a `forward` statement: the only difference is the fact that a `forward` statement defines a label visible to the environment and that the condition can be omitted (which means that it is implicitely $true$) in the case of an unconditional `forward`. We can therefore formally denote the abstractions $\mathcal{A}$ of those statements α as the set of tuples of a Boolean expression b_α and a set of label references L_α. So the set of abstractions $\mathcal{A}$ is given as

$$\mathcal{A} = \mathcal{B} \times \mathcal{P}(\mathcal{L})$$

where $\mathcal{B}$ is the set of all Boolean expressions and $\mathcal{P}(\mathcal{L})$ denotes the power set over the set of label references. The set of `guards` statements is therefore defined as $\mathcal{G} = \mathcal{A}$ and the set of `forward` statements is defined as $\mathcal{F} = \mathcal{ID} \times \mathcal{A}$, where $\mathcal{ID}$ represents its label. Given the abstraction $\alpha = (b_\alpha, L_\alpha)$, a label reference $l \in L_\alpha$ is called *resolved*, if it refers to a guard label inside a behavior instance. Moreover, α is *resolved* if all of its labels are *resolved*. We define $L^R_\alpha = \{l \in L_\alpha \mid l \textit{ resolved}\}$ and $L^U_\alpha = L_\alpha \setminus L^R_\alpha$ (unresolved). We recurse now on the instance tree. When starting from the leaf instances, i.e. instances that do not contain any further subinstances, we can safely assume that they contain only `guards` or `forward` statements referring directly to guard labels inside their behaviors, i.e. all of them are *resolved*. When considering a `guards` - or a `forward` - statement in a parent node $p = (b_p, L_p) \in \mathcal{A}$, L_p can either refer over L^U_p to its *resolved* `forward` statements in its children nodes F or directly over L^R_p to the assigned behavior instances. For a particular `guards` or `forward` statement within node p, we can determine the set of conditional forwards F_C, i.e. all of the referred forwards $\mathcal{F}$ that comprise a Boolean expression not equal to $true$, by defining $F_C = \{f \in \mathcal{F} \mid \pi_1(\pi_2(f)) \not\equiv true\}$. (Hereby, π_i is the projection function, yielding the i-th element of a tuple.) From all conditional forwards F_C we retrieve the set of Boolean expressions

$$B = \bigcup_{f \in F_C} \pi_1(\pi_2(f)).$$

Now we can generate the resulting set of expanded statements P' derived from p by considering all possible combinations of the set of Boolean expressions B:

$$P' = \bigcup_{B_t \in \mathcal{P}(B)} (b'(B_t), L'(B_t)) \ ,$$

where for each combination B_t a Boolean expression b' is constructed as a conjunction of the Boolean expression of the parent statement b_p, the elements of the corresponding combination of unnegated terms B_t and the (negated) elements of its complement $B \setminus B_t$

$$b'(B_t) = b_p \wedge \left(\bigwedge_{x \in B_t} x \right) \wedge \left(\bigwedge_{x \in B \setminus B_t} \neg x \right).$$

Moreover for each combination B_t, the corresponding label references L' have to be constructed as the union of the label references of the parent statement that were already resolved L^R_p, the label references of the unconditional forwards (i.e. $F \setminus F_C$) and the label references of those conditional forwards F_C that are "enabled" since their Boolean expression is in the unnegated set B_t:

$$\begin{aligned} L'(B_t) = \ & L^R_p \ \cup \\ & \left(\textstyle\bigcup_{f \in F \setminus F_C} \pi_2(\pi_2(f)) \right) \ \cup \\ & \left(\textstyle\bigcup_{f \in F_C : \pi_1(\pi_2(f)) \in B_t} \pi_2(\pi_2(f)) \right) \end{aligned}$$

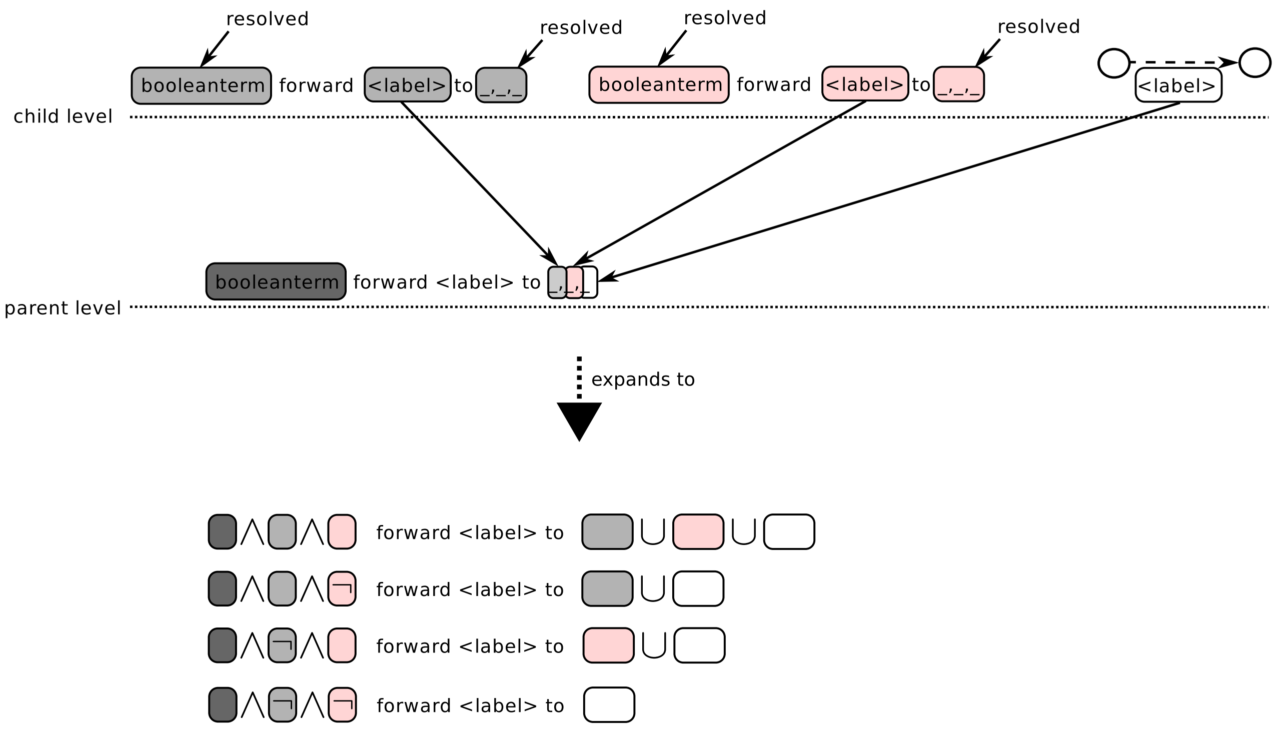

Figure 5: *Scheme of the* `forward` / `guards` *Expansion*

Fig. 5 illustrates the scheme described by these formulas. Note that for brevity we did not consider inside the given formulas that the label references and the Boolean expression literals namespace has to be suffixed by the instance over which they have have been resolved. After performing an expansion and obtaining P' as the transformed guards or `forward` statements, p can be substituted by the statements within P'. While performing the expansion over the tree structure of the model starting at the leaf instances, all associated guards - and `forward` statements will be expanded. When reaching the root node of the instance tree, all guards are expanded and therefore *resolved*, i.e. referring directly to the guard labels of the instantiated behaviors. In consequence, the resolved `forward` statements are not needed any more. In the running example the outcome is pretty straightforward since we do not have conditional forwards (i.e. the Boolean expression is implicitly $true$). For the network instance the two guard statements are resolved by the forwards associated to the link instances, resulting in the $LARES_{BASE}$ statements

```
(
  l[1].BLink.lfailed & l[2].BLink.lok
) guards  l[2].BLink.<swactive>

(
  (l[1].BLink.lfailed & l[2].BLink.lfailed) &
  l[3].BLink.lok
) guards  l[3].BLink.<swactive>
```

while for the system instance the guard remains the same, since `BSys.<systemfail>` is already a resolved label and consequently the whole statement is considered as being *resolved*:

```
(
  (
    p[1].BProcessor.pfailed | p[2].BProcessor.pfailed
  ) | (
    n.l[1].BLink.lfailed & n.l[2].BLink.lfailed &
    n.l[3].BLink.lfailed
  )
) guards  BSys.<systemfail>
```

4.2. Resolving Hierarchy and Behavior Transformation into SPA

To recall, a guards statement within a $LARES_{BASE}$ model evaluates the state of its related instance tree (which we denote as its generative part) and triggers certain guard labels within the behavior instances (its reactive part). The reactive part follows a max-sync semantics, which means that the maximum possible subset of the behavior instances will participate in the synchronization.

We illustrate this by a simple example:

```
A.a & B.b  guards  A.<x>, C.<y>
```

Here the generative part consists of only a single satisfying path i.e. $A.a \wedge B.b$. The reactive part consists of all possible combinations as shown in Table 1. (The last line of the table is provided only for the sake of completeness. Since no process is able to react, no synchronising transition needs to be generated in this case.) In the last column of the table, the table assigns to each case a unique identifier which will later be used

generative part	reactive part (max sync)	unique action label
$A.a \quad \wedge \quad B.b$	$A.\langle x\rangle \quad \wedge \quad C.\langle y\rangle$	$g0r0$
	$A.\langle x\rangle \quad \wedge \quad \neg C.\langle y\rangle$	$g0r1$
	$\neg A.\langle x\rangle \quad \wedge \quad C.\langle y\rangle$	$g0r2$
	$\neg A.\langle x\rangle \quad \wedge \quad \neg C.\langle y\rangle$	$g0r3$

Table 1: *Generative and Reactive Combinations*

generative augm. Literals	reactive augm. Literals
$((ns, gN, g0r0), A.a)$, $((ns, gN, g0r0), B.b)$	$((ns, gN, g0r0), \ A.\langle x\rangle)$, $((ns, gN, g0r0), \ C.\langle y\rangle)$
$((ns, gN, g0r1), A.a)$, $((ns, gN, g0r1), B.b)$	$((ns, gN, g0r1), \ A.\langle x\rangle)$, $((ns, gN, g0r1), \neg C.\langle y\rangle)$
$((ns, gN, g0r2), A.a)$, $((ns, gN, g0r2), B.b)$	$((ns, gN, g0r2), \neg A.\langle x\rangle)$, $((ns, gN, g0r2), \ C.\langle y\rangle)$
$((ns, gN, g0r3), A.a)$, $((ns, gN, g0r3), B.b)$	$((ns, gN, g0r3), \neg A.\langle x\rangle)$, $((ns, gN, g0r3), \neg C.\langle y\rangle)$

Table 2: *Augmented Generative and Reactive Combinations*

as part of an SPA action label.

In the sequel we explain the general case:

1. Since each `guards` statement describes an interaction of process instances, unique action labels have to be generated for each combination of its generative and reactive parts. The action label has to include the namespace $\mathcal{NS}$ of the module instance. This namespace is just a tuple of instance identifiers $\mathcal{ID}$, denoting the current module instance in the hierarchy.

2. The generative part comprises a Boolean expression over states, which is represented by disjunctive normal form (DNF) in the standard way. The generative part will be translated (in the process algebra domain) to a parallel composition with a defined synchronization set as follows: each minterm contributes to a different action label $gX \in \mathcal{L}_g$ over which the queried processes synchronize.

3. The reactive part comprises a list of guard label references, where all referenced behaviors synchronize, if possible. This is addressed by considering all subsets, where always at least one label has to imply a behavioral change. By considering all subsets individually (as exemplified in the above table), it is assured that no illegal choices are allowed. Each subset represents the remaining contribution $rY \in \mathcal{L}_r$ to the action label $gXrY$ that will be constructed.

A guard-combination identifier $g_{id} \in G_{id}$ is then a tuple $g_{id} = (ns, gN, gX, rY)$, where gN is the guard number within namespace ns. Each literal used in a combination of generative and reactive parts of a `guards` statement is augmented by its guard-combination identifier, i.e. tuples of the form $(g_{id}, l) \in G_{id} \times (\mathcal{L}_g \cup \mathcal{L}_r)$ are built (shown in Table 2). These augmented literals are all generated while traversing the model instances. Whenever `guards` statements are found, a unique action label identifier is constructed as stated above. As a consequence, for the above example we obtain Table 2. These tuples are aggregated and forwarded while traversing along the instance tree in the direction to its leaves. When reaching a particular behavior instance, a transformation function is applied. It is declared as follows:

$$t_B : \mathcal{NS} \times \mathcal{ID} \times B \times \mathcal{P}(\mathcal{L}_g) \times \mathcal{P}(\mathcal{L}_r) \to P$$

We denote $(ns, asgnId, b, L_g, L_r)$ as the argument tuple (with the namespace ns of the behavior instance, its assigned identifier $asgnId$, the behaviors definition b and the generative and reactive literals L_g and L_r) of that function. This has to be mapped into a sequential process $p \in P$ with assigned information about how to synchronize with its environment. The transformation of a behavior instance is then defined as follows:
By concatenating the namespace ns and the assigned identifier $asgnId$ a unique identifier for the resulting SPA process p is constructed. Next, the set of generative augmented literals L_g are partitioned into $L_g^{\not\neg}$ and $L_g^{\neg}$, considering whether the literal is a unnegated or negated one. The same is done for the reactive augmented literals, i.e. defining $L_r^{\not\neg}$ and $L_r^{\neg}$. For the transformation of guarded transitions we have to consider different cases w.r.t. the generative literals of a single guard-combination identifier:

1a) $\exists$ unnegated generative literals $\wedge$
$\exists$ negated generative literals,

1b) $\exists$ unnegated generative literals $\wedge$
$\nexists$ negated generative literals,

2) $\nexists$ unnegated generative literals $\wedge$
$\exists$ negated generative literals and

3) $\nexists$ unnegated generative literals $\wedge$
$\nexists$ negated generative literals.

Cases 1a) and 1b) can be treated uniformly, since case 1a) is included in case 1b). The reason is that a state is only associated to exactly one identifier that a literal can refer to. Likewise, we have to consider the cases concerning the reactive augmented literals of a single guard-combination identifier:

1a) $\exists$ unnegated reactive literals $\wedge$ $\exists$ negated reactive literals,

1b) $\exists$ unnegated reactive literals $\wedge$ $\nexists$ negated reactive literals,

2) $\nexists$ unnegated reactive literals $\wedge$ $\exists$ negated reactive literals and

3) $\nexists$ unnegated reactive literals $\wedge$ $\nexists$ negated reactive literals.

As before, the cases 1a) and 1b) can be treated uniformly since a transition is only associated to exactly one single guard label.

Thus, in combination there are 9 distinguishable cases to consider. Based on all cases stated above, sets of SPA transitions $T_{\{1,2,3\}^2}$ are constructed for a specific behavior instance:

$$T_B = \left\{ \bigcup_{x,y \in \{1,2,3\}^2} T_{xy}(g_{id}) \;\middle|\; g_{id} \in G_{id} \right\}$$

The family of functions T_{xy} generates those transitions of the SPA process p which are associated with the guard g_{id}. In addition, the local transitions of a behavior instance are generated and become part of the SPA process. Details of their implementation are beyond the scope of this paper. But apart from constructing p, the construction of the overall SPA model as a hierarchical composition of processes is an important issue. In this context, it is important to mention that almost all action labels derived from g_{id} have to synchronize with the environment of process p (the only exceptions are "self-affecting" guards where both the generative and the reactive parts refer to the same single behavior). This concept for composing the SPA structure via action synchronisation which we denote as $PACT$ (Process Algebra Composition Tuple) was already developed in [Riedl et al. (2010)] and has been refined since then. In this paper, we do not present the code of the resulting SPA model for our running example, but in the next section the analysis of that example is carried out.

5. ANALYSIS OF THE RUNNING EXAMPLE

For each model parameterization (e.g. in the case of our example model we can set the parameter `numLink` with different values) the transformations can be applied, resulting in a CASPA SPA model. The workflow chain as given in Fig. 8 allows CASPA to be executed with the generated CASPA model code and certain additional arguments, stating for instance whether to perform steady-state or transient analysis or whether to dump the resulting statespace. With model parameter `numLink` set to 3 the resulting statespace is too large (57 reachable states after elimination) to be depicted here. Therefore, we provide Fig. 6 as an example for the statespace when setting `numLink` to 1. As one can see, the composed states are tuples with one entry for each instantiated behavior. Each state is composed of the four states of the behavior instances, i.e. the system behavior instance `FTN_BSYS`, the single link behavior instance `FTN_n_l[1]_BLink` and the two processor behavior instances `FTN_p[1]_BProcessor` and `FTN_p[2]_BProcessor`. For example, the topmost state, i.e. the initial state, is composed of the states `(sok,lactive,pok,pok)`. In Fig. 7 the transient probability of being in a composed state where a system hazard has already taken place is calculated by CASPA for different parameter sets. Note that each step, starting from the LARES specification to the quantitative evaluation of the SPA model, is performed automatically. Also the statespace figure and the diagram with the measure of interest can be directly generated by our tool chain. Each curve within Fig. 7 corresponds to a specific configuration, i.e. a specific combination of values for the link failure rate and the number of links. It requires the workflow to be performed for each configuration and subsequently be analysed by CASPA as many times as data-points for the curves are needed. Due to the fully-automated process no cumbersome manual work has to be done any more.

6. FURTHER WORK

In Fig. 8, different transformation workflows and their relation to each other are shown. While in [Gouberman et al. (2012)], the LARES modelling and analysis environment implementation is presented in general, this paper focused on the transformation workflow starting with a LARES model, which is expanded to a $LARES_{BASE}$ model and finally transformed into a CASPA SPA model that can be analyzed using CASPA. Since $LARES$ aims at different solvers as backends, also other transformations are provided. Hereby it is important to maintain the same behavior over all transformations:

Validation
When developing a new transformation workflow or modifying existing transformation steps, the resulting behavior has to remain always equivalent. This is assured by first eliminating all vanishing states resulting from the immediate transitions and then applying a strong Markovian bisimulation algorithm [Hermanns and Siegle (1999)]. The eliminated Markovian labeled transition systems obtained from both, performing CASPA reachability analysis and a reachability analysis

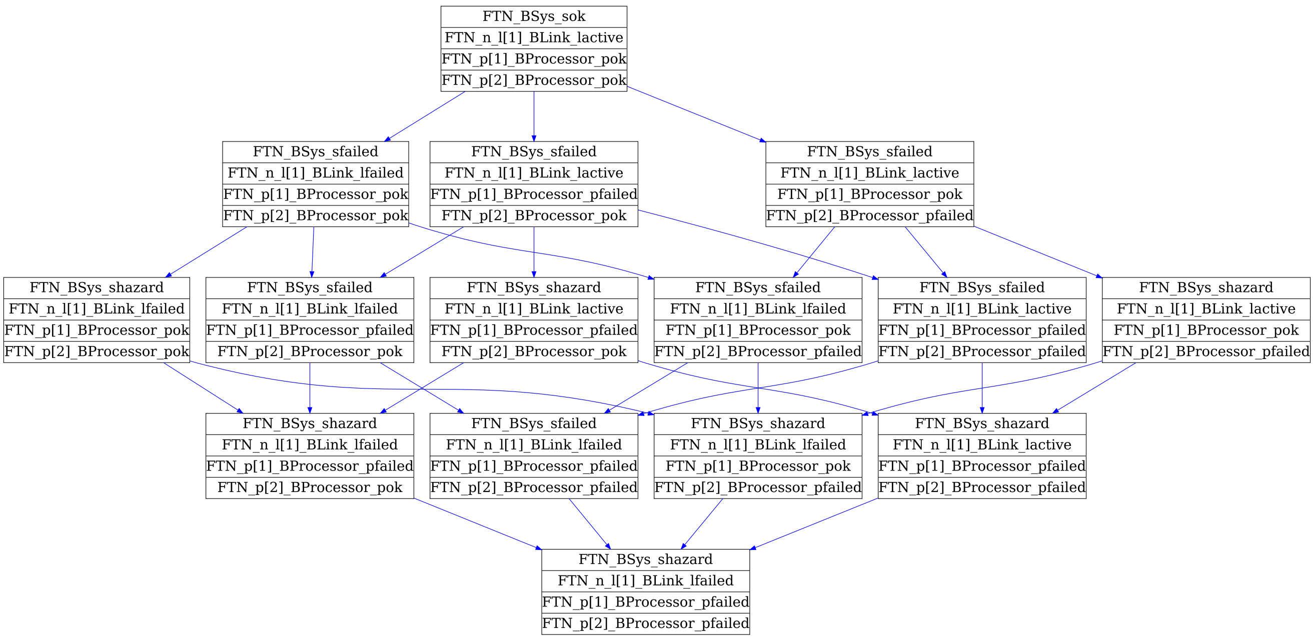

Figure 6: *Resulting Statespace for* `numLink`*=1*

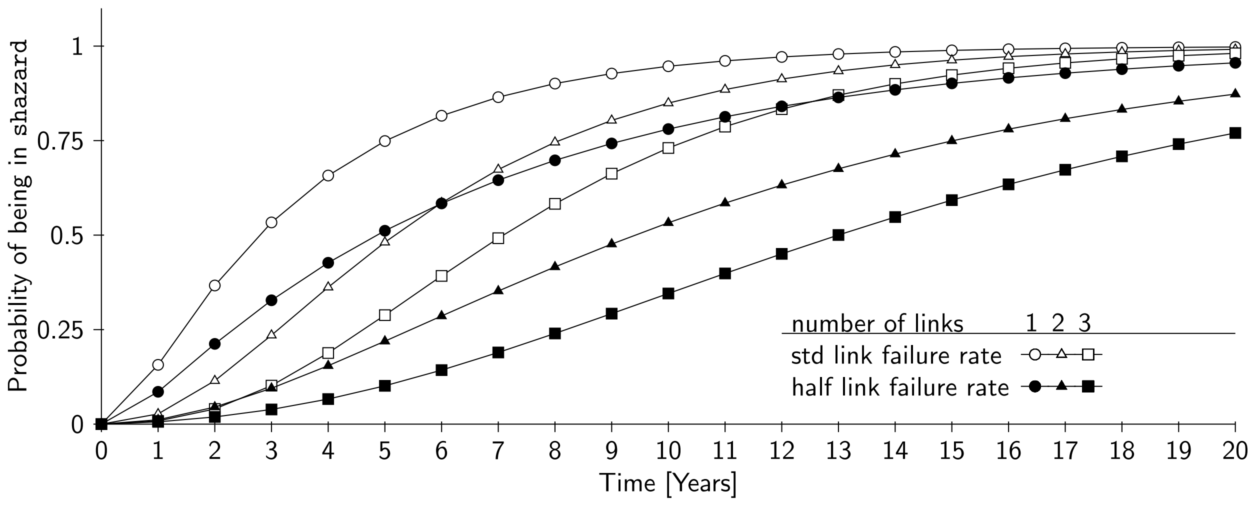

Figure 7: *Transient Probabilities*

based on a planar representation of $LARES$ (i.e. $LARES_{FLAT}$, see Fig. 8), serve as input to the bisimulation algorithm, which finally decides whether both transition systems (TRA) are equivalent ($\sim$) with respect to strong Markovian bisimulation. As a long-term goal, a proof on the equivalence between the LTS and the SPA transformation semantics could be conducted, stating that both transformations yield the same CTMC for each possible input model. However, this proof has not yet been carried out. Apart from a formal proof of the theoretical correctness of the transformation semantics, correctness of its implementation is hard to prove, and therefore we assume the implementation to be sound by testing it on a number of models with different level of complexity.

State-Space Explosion
Reduction techniques on the language-level are currently not supported. However, analysis of the structure and interaction between instances could help to determine independent subcomponents or instances which are behaviorally equivalent with respect to their environment. Independent subcomponents can be solved independently and their results reused within the upper-model. It is also possible to aggregate a large number of equivalently behaving instances by analyzing the relative number of components being in a certain state (e.g. by mean-field analysis [Le Boudec et al. (2007)]).

Petri Net Transformation
The $LARES_{FLAT}$ planar model has been developed as a by-product of an additional transformation workflow that has been realized to obtain stochastic Petri net models. There, the $eDSPN$ formalism of [Zimmermann et al. (2006)] has been used as a target formalism to allow the use of TimeNet as the corresponding solver. Additionally, a serialization has been defined on the $eDSPN$ formalism to provide an output conforming the $SPNP$ syntax [Ciardo, Muppala and Trivedi (1989)].

Editors
In addition to an already existing textual $LARES$ editor for the Eclipse environment[3] that provides the modeler with auto-completion and syntax highlighting, a graphical editor is currently under development.

7. CONCLUSION/OUTLOOK

Notwithstanding that the paper did not provide a fine-grained formalization of the transformation semantics, an insight has been given on how the transformation workflow from $LARES$ into CASPA SPA is formalized and implemented. As already addressed, the resulting framework comprises of a number of transformations, algorithms for validation of the transformations and routines to apply the associated solvers. Mechanisms have been established to easily modify or complement the transformation workflows.

In the near future we plan to integrate the framework within the textual/graphical editor to provide a suitable IDE for LARES.

Acknowledgments.
Special thanks to Johann Schuster for helping to improve the comprehensibility of the transformation notation, and to David Jansen for his sharp eyes on the running example. The authors are also grateful to the anonymous reviewers for their valuable comments. Further, we would like to thank Deutsche Forschungsgemeinschaft (DFG) who supported this work under grants SI 710/7-1 and for partial support by DFG/NWO Bilateral Research Programme ROCKS.

8. REFERENCES

Bachmann, J., Riedl, M., Schuster J., and M. Siegle.(2009) *An Efficient Symbolic Elimination Algorithm for the Stochastic Process Algebra Tool CASPA.* In *SOFSEM '09: Proceedings of the 35th Conference on Current Trends in Theory and Practice of Computer Science*, pages 485–496, Berlin, Heidelberg, 2009. Springer LNCS 5404.

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

M. Bozzano, A. Cimatti, J.-P. Katoen, V. Y. Nguyen, T. Noll, and M. Roveri.(2011) *Safety, Dependability and Performance Analysis of Extended AADL Models. The Computer Journal*, 54(5):754–775, 2011.

G. Ciardo, J. Muppala, and K. Trivedi.(1989) *SPNP: Stochastic Petri Net Package.* In *Proceedings of the Third International Workshop on Petri Nets and Performance Models*, pages 142 –151, Dec 1989.

A. Gouberman, M. Riedl, J. Schuster, and M. Siegle.(2012) *A modelling and analysis environment for lares.* In Jens B. Schmitt, editor, *MMB/DFT*, volume 7201 of *Lecture Notes in Computer Science*, pages 244–248. Springer, 2012.

A. Gouberman, M. Riedl, J. Schuster, M. Siegle, and M. Walter.(2009) *LARES - A Novel Approach for Describing System Reconfigurability in Dependability Models of Fault-Tolerant Systems.* In *ESREL '09: Proceedings of the European Safety and Reliability Conference*, pages 153–160. Taylor & Francis Ltd., 2009.

V. Grassi, R. Mirandola, E. Randazzo, and A. Sabetta.(2008) *KLAPER: An Intermediate Language for Model-Driven Predictive Analysis of Performance and Reliability.* In *The Common Component Modeling Example*, volume 5153 of *Lecture Notes in Computer*

[3]Eclipse IDE: `http://www.eclipse.org`

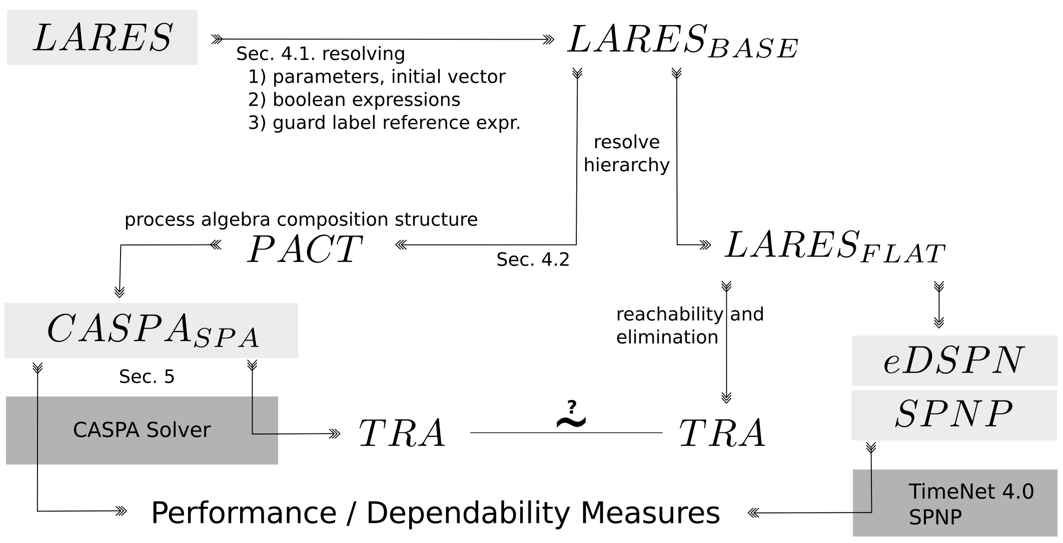
LARES
Sec. 4.1. resolving
1) parameters, initial vector
2) boolean expressions
3) guard label reference expr.
LARES_BASE
resolve hierarchy
process algebra composition structure
PACT
Sec. 4.2
LARES_FLAT
CASPA_SPA
Sec. 5
reachability and elimination
eDSPN
SPNP
CASPA Solver
TRA
TRA
TimeNet 4.0
SPNP
Performance / Dependability Measures

Figure 8: *Transformation Workflows*

Science, pages 327–356. Springer Berlin / Heidelberg, 2008.

H. Hermanns and M. Siegle.(1999) *Bisimulation Algorithms for Stochastic Process Algebras and Their BDD-Based Implementation.* In *ARTS'99*, pages 244–264, 1999.

M. Kuntz, M. Siegle, and E. Werner.(2004) *Symbolic Performance and Dependability Evaluation with the Tool CASPA*. In *Applying Formal Methods: Testing, Performance and M/E Commerce: FORTE 2004 Workshops, European Performance Engineering Workshop*, pages 293–307. Springer, LNCS 3236, 2004.

M. Lê and M. Walter.(2011) *Considering Dependent Components in the Terminal Pair Reliability Problem*. In *Proc. of The Second Workshop and Tool Session on DYnamic Aspects in DEpendability Models for Fault-Tolerant Systems (DYADEM-FTS)*, pages 415-422, IEEE, 2011.

J.-Y. Le Boudec, D. McDonald and J. Mundinger.(2007) *A Generic Mean Field Convergence Result for Systems of Interacting Objects*. In *Proc. of Fourth Int. Conf. on the Quantitative Evaluation of Systems (QEST)*, pages 3–18, IEEE, 2007.

M. Riedl, J. Schuster, M. Siegle, M. Blum, and F. Schiller.(2010) *Dependability Model Transformation – A Stochastic Process Algebra Semantics for ZuverSicht Models*. In *Reliability, Risk and Saftey: Back to the Future. Proceedings of the European Safety and Reliability Conference (ESREL 2010)*, pages 932–940. Taylor & Francis Group, London, 2010. ISBN 978-0-415-60427-7.

M. Walter.(2011) *Simple Non-Markovian Models for Complex Repair and Maintenance Strategies with LARES+*. In *Advances in Safety, Reliability and Risk Management (Proc. of ESREL 2011)*, pages 962–969. Taylor & Francis Group, London, 2011.

M. Walter.(2011) *Stepwise Refinement of Complex Dependability Models Using LARES+*. In *Sixth International Conference on Availability, Reliability and Security (ARES 2011)*, pages 436–441, 2011.

M. Walter and M. Lê.(2011) *Clear and Concise Models for Fault-Tolerant Systems with Limited Repair using the Modeling Paradigm LARES+*. In *19th AR2TS Advances in Risk and Reliability Technology Symposium*, pages 310–321, 2011.

A. Zimmermann, M. Knoke, A. Huck, and G. Hommel.(2006) Towards version 4.0 of TimeNET. In *MMB*, pages 473–476. VDE Verlag VDE Verlag, 2006.