# A Modelling and Analysis Environment for *LARES*

Alexander Gouberman, Martin Riedl, Johann Schuster, and Markus Siegle

Institut für Technische Informatik,
Universität der Bundeswehr München,
{firstname.lastname}@unibw.de

**Abstract.** This paper presents a toolset for modelling and analysis of fault tolerant systems based on *LARES* (*LA*nguage for *RE*configurable *S*ystems), an easy-to-learn formalism that allows its users to describe system structure, dynamic failure and repair behaviour.

## 1 Introduction

Modern societies rely on the correct and timely functioning of complex systems, e.g. in the communication, transportation or energy sectors, which makes it extremely important that we understand and are able to improve the dependability of these systems.

In this short paper, we present the *LARES* toolset, a new software library which supports dependability modelling and analysis. It is designed to be integrated into the development cycle of modern IT-based systems, since it offers transformations from / to various other modelling languages. At the core of the toolset is the domain specific *LARES* language, first presented in [3] and since then employed in several modelling case studies, e.g. [4]. This language allows the user to specify in a clear and concise way all aspects of a system which are relevant for its dependability. Apart from basic combinatorial failure conditions, complex error situations such as error propagation or common cause failures, arbitrary redundancy structures and different repair strategies can be modelled in a strictly modular and hierarchical fashion.

The *LARES* toolset, presented here for the first time, does not contain its own analysis engine, but relies on external tools for qualitative and quantitative analysis. For this purpose, transformations to various target formalisms such as stochastic process algebra (SPA), stochastic Petri nets (SPN) or some simulation language need to be provided. Some of these have been implemented already, in particular the present paper focuses on the transformation to the SPA tool *CASPA* [1] which consists of several steps, as discussed below.

## 2 The *LARES* Modelling Language

For explaining how modularity and hierarchy can be modelled in *LARES*, a small example of a fault tolerant system is given in Fig. 1. The system `RedComp` consists of two components `iC` and `iSer` whose descriptions are captured inside the (abstract) module definitions `mComp` and `mSerialC`. The module `mComp` includes a behaviour `bComp` which defines the state space for each instance of this module. Concretely, the lifetime of the component `iC` is phase-type distributed with an initial state `Good`, an erroneous state `Error` and an absorbing state `Failed`

indicating a failure of the component. From an erroneous state the component can recover to the state `Good` with rate `0.2` or finally fail with rate `0.3`. The component `iSer` consists of two subcomponents `iComp[1]` and `iComp[2]` which have the same lifetime distribution as `iC` since they are instantiated from the same abstract module `mComp`. Inside a module, conditions can be defined which capture information about the states of the subcomponents, e.g. the component `iSer` fails (defined in the condition `failed`) if at least one of its subcomponents fails. This condition is used in order to lift state information towards the system level in the structural model hierarchy. On the level of the system, if both components `iC` and `iSer` fail, the event `intEvent` is generated which immediately forces the system to fail. Beside its two components, the system has an additional error behaviour `bErr`: it can fail either if the guard `<intEvent>` is triggered (if both components fail) or due to an external event after an exponentially distributed lifetime with rate `0.082`. Since the system does not provide an explicit start state for `bErr`, the first occurring state is used.

```
Behavior bErr(externalEventRate){
   Transitions from Good
     if <true> → Failed, delay exponential externalEventRate
     if <intEvent> → Failed
}
System RedComp : bErr(externalEventRate=0.082) {
  Behavior bComp {
     Transitions from Good if <true> → Error, delay
          exponential 0.1
     Transitions from Error
           if <true> → Good, delay exponential 0.2
           if <true> → Failed, delay exponential 0.3
  }
  Module mComp : bComp {
    Condition failed = bComp.Failed
    Initial init = bComp.Good
  }
  Module mSerialC {
    expand (i in {1 .. 2}) { Instance iComp[i] of mComp }
    Condition failed = OR[i in {1 .. 2}] iComp[i].failed
  }
  Instance iSer of mSerialC
  Instance iC of mComp
  iSer.failed & iC.failed guards bErr.<intEvent>
}
```

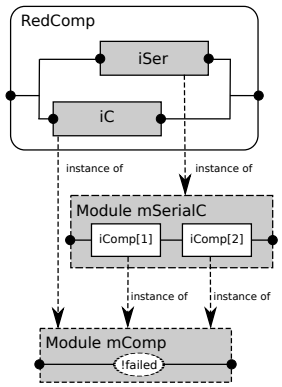


**Fig. 1.** Fault Tolerant Example Model

We restrict in this model to exponentially distributed time delays (by using phase-type distributions) in order to be able to analyze it exactly with Markov chain methods. The *LARES* language also allows for general distributions, in which case the model needs to be analyzed with simulative methods.

## 3 Tooling and Transformations

Fig. 2 shows the tools used (dark grey filled rectangles) and the different transformation steps into specific formalisms such as SPN or SPA. A *LARES* model can be specified using an editor component that supports the user with syntax highlighting, auto-completion and of course syntactical and partial semantical evaluation of correctness of the model. There is a multistage transformation that first evaluates all expressions, thereby expanding all parameter dependent statements, then resolving the hierarchical indirections over `Condition` statements leading to resolved logical propositions on states used inside the `guards` statements, and finally resolving the indirections of the guard label references over the hierarchy. These steps yield the $LARES_{BASE}$ subset from which two different types of transformation are supported: Firstly, the transformation into a stochas-

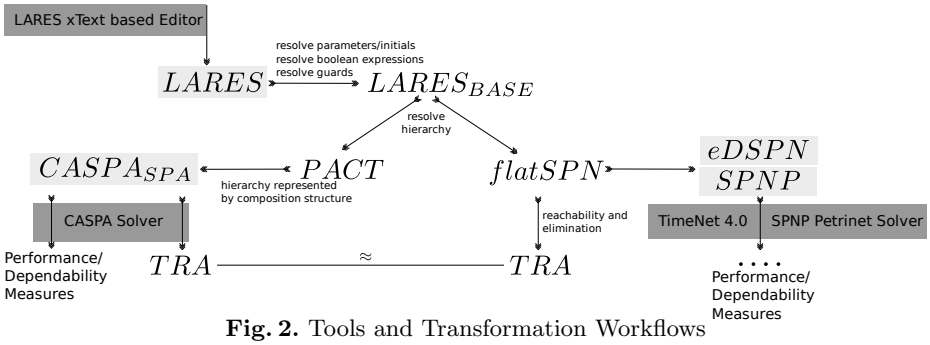


**Fig. 2.** Tools and Transformation Workflows

tic process algebra, where synchronisation is used to handle the different cases, i.e.: Does a certain state of a component contribute to a valid generative composed state? Which reaction, based on the transitions referred to by a guard label reference, could follow? From this, a sequential stochastic process is generated and integrated by synchronisation information, to specify which transitions have to interact with their environment. While decomposing the *LARES* hierarchy, the SPA composition tree *PACT* is built, consisting of the individual processes at its leaves and their subsequently generated intermediate composition nodes and their synchronisation sets. The generated composition structure is then decomposed, resulting in an SPA specification which can be analysed using the CASPA solver. Secondly, the hierarchy can be resolved leading to a stochastic Petri net (i.e. *flatSPN*) with enabling conditions for transitions. Based on this model, either a reachability analysis can be performed, resulting in a transition system (*TRA* file), or alternatively, it can be translated to an *eDSPN* model to apply TimeNet [5] or to an *SPNP* model [2].

Applying the transformation workflow into CASPA SPA, we can automatically derive certain measures. Fig. 3 shows a curve obtained by transient analysis of our example for a number of timepoints for a given state measure defined on the `Failed` state of the system error model. Since the model has an absorbing state (i.e. components cannot recover), the curve converges to one.

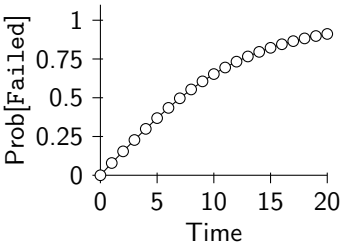


**Fig. 3.** Analysis Example

## 4 Implementation Details

A textual editor for *LARES* has been implemented using Eclipse TMF xText, in order to provide a suitable modelling environment to the user (a graphical editor is under development). xText provides an easy way to implement domain specific languages such as *LARES*. Moreover, we chose the Scala language to implement the transformations into the solver formalisms. It helped a lot to apply its functional and OO concepts and therefore obtain a code which is close to the mathematical formalization. The abstract syntax tree definition has been built using algebraic data types (i.e. using Scala case classes). Furthermore, the

concrete syntax has been implemented using Scala parser combinators. By applying the root parser to a *LARES* specification, the abstract syntax tree is created, i.e. a model is loaded. Next, the transformation has to be done. Classically, the visitor pattern is applied to assure a separation between the abstract syntax tree implementation and the transformation code. As the tree consists of algebraic data types, one can use Scala pattern matching for decomposition instead of traversing the tree with the visitor concept, retaining the separation of the abstract syntax tree implementation and transformation code. The *LARES* instance tree is traversed multiple times to apply all transformation steps described above. After thus obtaining the intermediate *PACT* representation, a composition routine is executed, resulting in the SPA model. Lastly, a simple model-to-text transformation is done on the SPA model whose result is subsequently handed over to the *CASPA* solver. Transformation validation can be performed, which helps to assure the correctness of the transformation workflows even when applying changes to the code. This is done using ScalaCheck, a unit testing library that allows to define a number of testcases. When performing a test, two different workflows are executed resulting in two transition systems on which a comparison (i.e. bisimulation equivalence) is performed. ScalaCheck finally states whether the two transition systems are bisimilar or not, or if an error occurred during the transformation. Whenever a code change is done, all testcases are evaluated again.

## 5 Conclusion

We have presented an environment in which *LARES* dependability models can be specified and analysed. We introduced the *LARES* formalism by example and gave insight into the different transformation steps which lead to an analysable model. Moreover, we also briefly touched transformation validation. Readers interested in the toolset are asked to contact the authors.

**Acknowledgments.** 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.

## References

1. J. Bachmann, M. Riedl, J. Schuster, and M. Siegle. An Efficient Symbolic Elimination Algorithm for the Stochastic Process Algebra Tool CASPA. In *SOFSEM '09: Proc. of the 35th Conf. on Current Trends in Theory and Practice of Computer Science*, pages 485–496, Berlin, Heidelberg, 2009. Springer LNCS 5404.
2. G. Ciardo, J. Muppala, and K. Trivedi. SPNP: Stochastic Petri Net Package. In *Proc. of the Third Int. Workshop on Petri Nets and Performance Models*, pages 142 –151, Dec 1989.
3. A. Gouberman, M. Riedl, J. Schuster, M. Siegle, and M. Walter. LARES - A Novel Approach for Describing System Reconfigurability in Dependability Models of Fault-Tolerant Systems. In *ESREL '09: Proc. of the European Safety and Reliability Conf.*, pages 153–160. Taylor & Francis Ltd., 2009.

4. M. Walter and M. Lê. 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.
5. A. Zimmermann, M. Knoke, A. Huck, and G. Hommel. Towards version 4.0 of TimeNET. In Reinhard German and Armin Heindl, editors, *MMB*, pages 473–476. VDE Verlag VDE Verlag, 2006.