An IDE for the LARES Toolset

Alexander Gouberman[1], Christophe Grand[2], Martin Riedl[1], and Markus Siegle[1]

[1] Institut für Technische Informatik,
Universität der Bundeswehr München,
Werner-Heisenberg-Weg 39, 85579 Neubiberg, Germany
firstname.lastname@unibw.de

[2] Pôle SPID,
ENSTA-Bretagne,
2 rue François Verny, 29200 Brest, France
christophe.grand@ensta-bretagne.fr

Abstract. In order to support the editing, validation and analysis of LARES dependability model specifications, a textual editor and a graphical user interface for performing experiments have been developed. In collaboration with the LARES toolset library they serve as an Integrated Development Environment (IDE) based on the Eclipse framework. The paper first introduces the features of the LARES language by means of a hysteresis model taken from the literature. It then describes the textual Editor Plugin. Beyond standard features such as syntax highlighting and code completion, it emphasises syntactical and semantic validation capabilities. Subsequently, the View Plugin component is presented, that is used to perform the experiments and to gather the analysis results from the solvers. The current state of development of a graphical Editor Plugin and other features of the LARES IDE are also addressed.

1 Introduction

LARES (LAnguage for REconfigurable Systems) is a language and toolset for modelling the dynamic behaviour of systems, with a focus on dependability, fault-tolerance and reconfigurability. Previous papers about LARES focussed on the expressiveness of the modelling language [13, 14], its semantics [23, 14] and its transformation to evaluation formalisms such as Stochastic Process Algebra (SPA) or Stochastic Petri Nets (SPN) [12].
The present paper is the first one in which the LARES toolset is described from the user's point of view. We present an Integrated Development Environment (IDE), consisting of a sophisticated textual Editor Plugin as well as a View Plugin which serve as a comfortable user interface during all phases of model specification, validation, quantitative analysis and result presentation. Fig. 1 gives an overview of the LARES IDE, comprising the editor and the analysis view component, both Eclipse-based plugins, and the LARES library. When the

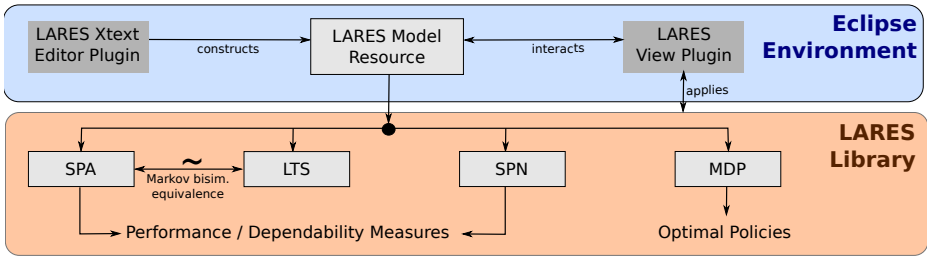


Fig. 1. The Eclipse-based LARES IDE

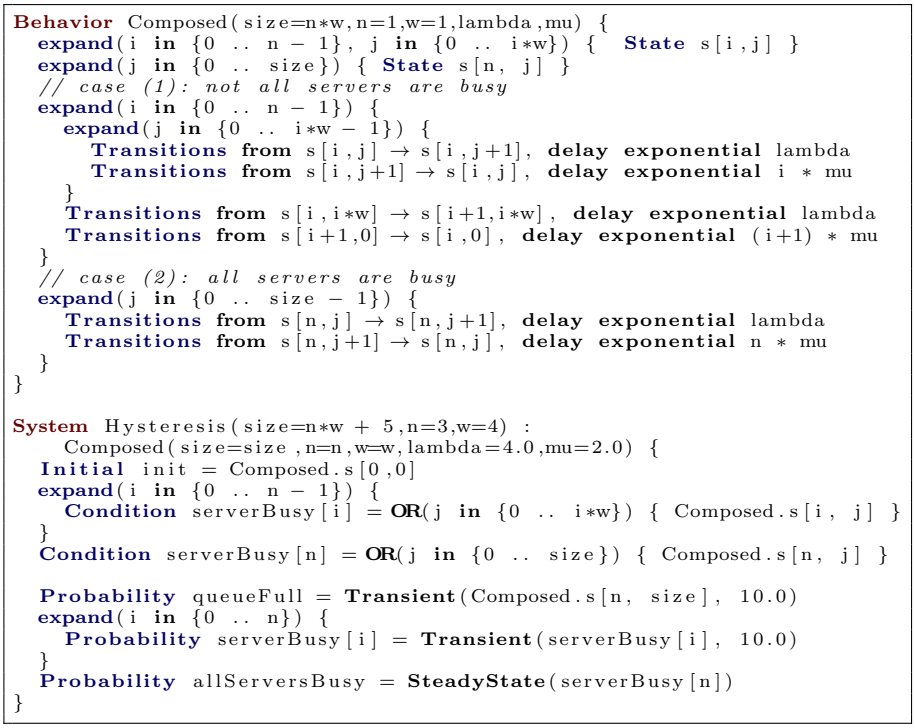
active page inside the Eclipse instance is a LARES specification (indicated by the filename extension *lrs*), the textual Editor Plugin becomes active and performs a validation of the model w.r.t. the LARES metamodel. The View Plugin parses the model, extracts the defined measures and constructs and visualises the instance graph (which depicts the model's structure in terms of submodel instances). The View Plugin also allows the user to perform additional checks (e.g. for the presence of deadlocks or unsafe states), to investigate the state space and to visualize analysis results. For interacting with the specified model, e.g. to initiate an analysis, the LARES library is applied. That library implements all aspects related to the language and its transformation to the target formalisms where the actual analysis is carried out. Beside the already mentioned SPA and SPN target formalisms we also support flat Labelled Transition Systems (LTS) and Markov Decision Processes (MDP). The library also interfaces with the solvers, thereby abstracting from their specific interfaces. The LARES toolset is available from [11]. Among related approaches we mention the COMPASS project with the hierarchic language SLIM [9] that applies the powerful nuSMV solver [10], MoDeST [17] as a very concise language to describe non-hierarchic systems applying numerous tools for analysis, and the very mature AltaRica language [21] that lately also includes some extensions to enable non-functional analysis. Compared to these approaches, LARES focusses on hierarchic systems with complex interaction patterns. Beyond triggering reactions based on the current state, LARES offers flexible types of synchronisation by means of reactive expressions which take the combinatorics of subsequent behaviours of each component into account.

The paper is structured as follows: Sec. 2 introduces the LARES language by means of a non-trivial example taken from the recent literature. It presents two approaches, representing different degrees of modularity of how to model the system in a more or less modular fashion. Sec. 3 details the textual Editor Plugin which – beyond syntax highlighting and code-completion – offers advanced syntactical validation, and also partial semantic validation. Sec. 4 presents the analysis environment and its capabilities to perform calculations and visualise the analysis results. Sec. 5 briefly sketches our current efforts in developing a graphical Editor Plugin and the View Plugin, after which the paper concludes with Sec. 6.

2 Compositional Modelling with LARES by Example

For describing the modelling capabilities of LARES, we consider the "multiple parallel hysteresis" queueing model from [20]. It is a model for load-dependent power-saving operation, where the activation and deactivation of servers occurs at different levels, leading to a hysteresis-like behaviour. The model is represented as a CTMC with states $(i, j) \in \mathbb{N} \times \mathbb{N}$, where $0 \leq i \leq n$ denotes the number of servers running in parallel ($n - i$ servers are idle) and $0 \leq j \leq size$ denotes the number of jobs in the system. Turning on an idle server can be problematic (e.g. causes costs or can lead to server breakdowns). For this reason, in order to bound the number of waiting jobs, a server is activated only if the number of jobs in the queue exceeds a certain threshold. In particular, the i-th server is activated if $j = w^{(i)}$ and deactivated only if the queue is empty ($j = 0$), which leads to a hysteresis (see Fig. 11(b)).

In order to specify this model in LARES, we first briefly outline the LARES language. A LARES model consists of Behavior and Module definitions. A Behavior represents (one dimension of) the state space of a system component, in which transitions can be guarded by a guard label and either delayed by an exponential distribution or triggered immediately by a weighted discrete distribution. A Module can instantiate Behaviors or other Modules, thus providing a hierarchy of instantiations (instance tree), in which the Module representing the root instance is specified by a System definition. Furthermore, in a Module definition, the guarded transitions of instantiated Behaviors can be triggered by a guards statement, dependent on assertions over states. This triggering can be either direct or by interaction with other guard labels in form of synchronisation or choice. LARES supports three types of synchronous interactions: sync (all addressed guard labels have to be provided to perform a synchronized transition), maxsync (all transitions which offer the guard labels will take place) and choose (if only a single guard label is offered then the transition will take place). Moreover, inside a Module definition one can specify further Behavior and Module definitions, Instance statements for instantiating Modules, and Condition statements representing logical expressions over states (or other Conditions). forward statements are like guards statements, but comprise additionally a forward label which may be triggered externally (producing an information flow towards all Behavior instances). Initial statements define an initial state configuration and Probability statements specify desired transient or steady state measures. Note that due to the instance hierarchy there are visibility constraints on states/conditions and guard/forward labels. Thus, Condition statements can be used in order to lift state assertions from Behavior instances towards a Module, and forward statements to propagate triggering events from a Module towards Behaviors. For increasing the modelling flexibility, Behaviors and Modules can be parametrised and expand statements can be defined, which define a shortcut for symmetric statements.
Due to the described expressiveness, there are several ways how to specify the hysteresis model with LARES. For the purpose of this paper, we present two



```
1  Behavior Composed(size=n*w,n=1,w=1,lambda,mu) {
     expand(i in {0 .. n - 1}, j in {0 .. i*w}) {  State s[i,j] }
     expand(j in {0 .. size}) { State s[n, j] }
     // case (1): not all servers are busy
     expand(i in {0 .. n - 1}) {
6      expand(j in {0 .. i*w - 1}) {
         Transitions from s[i,j] -> s[i,j+1], delay exponential lambda
         Transitions from s[i,j+1] -> s[i,j], delay exponential i * mu
       }
       Transitions from s[i,i*w] -> s[i+1,i*w], delay exponential lambda
11     Transitions from s[i+1,0] -> s[i,0], delay exponential (i+1) * mu
     }
     // case (2): all servers are busy
     expand(j in {0 .. size - 1}) {
       Transitions from s[n,j] -> s[n,j+1], delay exponential lambda
16     Transitions from s[n,j+1] -> s[n,j], delay exponential n * mu
     }
   }

   System Hysteresis(size=n*w + 5,n=3,w=4) :
21     Composed(size=size,n=n,w=w,lambda=4.0,mu=2.0) {
     Initial init = Composed.s[0,0]
     expand(i in {0 .. n - 1}) {
       Condition serverBusy[i] = OR(j in {0 .. i*w}) { Composed.s[i, j] }
     }
26   Condition serverBusy[n] = OR(j in {0 .. size}) { Composed.s[n, j] }

     Probability queueFull = Transient(Composed.s[n, size], 10.0)
     expand(i in {0 .. n}) {
       Probability serverBusy[i] = Transient(serverBusy[i], 10.0)
31   }
     Probability allServersBusy = SteadyState(serverBusy[n])
   }
```

Fig. 2. Planar LARES specification of a multiple parallel hysteresis model with queue length `size`, `n` servers and equidistant hysteresis widths $w^{(i)} = i \cdot \mathtt{w}$.

LARES specifications given in Fig. 2 and 3. Fig. 2 describes the whole hysteresis model in a planar way by defining a single Behavior `Composed` with parameters for the number `n` of servers (with default value `1`), the width `w` (s.t. the hysteresis widths are given by $w^{(i)} = i \cdot \mathtt{w}$), the `size` of the queue (set by default to `n*w`), and rates `lambda` and `mu` for the arrival and service processes (without default values). The first two expand statements (lines 2 .. 3) define the running indices `i` and `j` in order to declare all the states `s[i,j]`. The ranges for the indices are dependent on the values of other parameters. The expand statements (lines 5 .. 17) define all the necessary exponential transitions dependent on the two cases as specified in the comments. Note that expand statements can also be nested. The Behavior definition is instantiated in the System definition `Hysteresis` which defines its own parameters and sets (resp. overwrites) the parameters of the Behavior (line 21). Since the whole hysteresis model is specified in a single Behavior, there is no need to define any guard labels for the transitions. In the System we first set the initial state to `s[0,0]` and define the measure `queueFull` which computes the transient probability at $t = 10$ for the state `s[n,size]` of the Behavior instance `Composed` (line 28). Furthermore, in order to analyze the number of running servers, we specify the conditions `serverBusy[i]` which rep-

```
Behavior Queue(size=1, lambda) {
  expand(i in {0 .. size}) { State s[i] }
  expand(i in {0 .. size-1}) {
    Transitions from s[i] → s[i+1], delay exponential lambda
  }
  expand(i in {1 .. size}) {
    Transitions from s[i] if ⟨deq⟩ → s[i-1]
  }
}

Behavior Service(n=1, mu) {
  expand(i in {0 .. n}) { State active[i] }
  expand(i in {0 .. n-1}) {
    Transitions from active[i] if ⟨activateNext⟩ → active[i+1]
  }
  expand(i in {1 .. n}) {
    Transitions from active[i]
      if ⟨deactivate⟩ → active[i-1], delay exponential i * mu
      if ⟨process⟩ → active[i], delay exponential i * mu
  }
}

System Hysteresis(size=n*w + 5, n=3, w=4) :
          Q ← Queue(size=size, lambda=4.0),
          S ← Service(n=n, mu=2.0) {
  Initial init = Q.s[0], S.active[0]
  expand(i in {0 .. n-1}) {
    Condition switch[i] = Q.s[1 + i * w]
    switch[i] & S.active[i] guards sync { S.⟨activateNext⟩, Q.⟨deq⟩ }
  }
  !S.active[0] guards sync { S.⟨process⟩, Q.⟨deq⟩ }
  Q.s[0] guards S.⟨deactive⟩

  Probability queueFull = Transient(Q.s[size], 10.0)
  expand(i in {0 .. n}) {
    Probability serverBusy[i] = Transient(S.active[i], 10.0)
  }
  Probability allServersBusy = SteadyState(S.active[n])
}
```

Fig. 3. Towards greater modularity: Splitting the arrival and service process into different Behaviors.

resent all states in which exactly i servers are active (lines 23 .. 26). For this reason, we abstract of the number of jobs in the queue by disjunction.

As one can see, the planar representation of the whole hysteresis model can be difficult to understand and may be prone to errors (e.g. consistency regarding the index values and ranges). For this reason, Fig. 3 shows an alternative LARES specification of the same system by splitting the model into several parts: a Behavior for the `Queue` and a Behavior for the `Service` process. The queue is responsible for the arrival process (line 4) and provides for interaction a guard label `<deq>` in order to dequeue a job (line 7). Note that these guarded transitions do not provide any distribution type. This means that they act passively if a synchronisation is desired. The service process defines the states `active[i]` in order to denote i running servers. An additional server can be immediately activated if the guard label `<activateNext>` is triggered (line 14). Furthermore, a server can be deactivated, which takes some time in order to complete the

job first (line 18). By triggering the `<process>` guard label, the server remains active after service completion (line 19).
The System definition instantiates both behaviours with the names `Q` resp. `S` (lines 24 .. 25). The Condition `switch[i]` (line 28) provides information about states when an activation of a further server can take place, i.e. if a job enters the queue s.t. the hysteresis width $w^{(i)}$ is exceeded. In this case the guards statement (line 29) synchronously activates a further server and dequeues a job from the queue, but only if there are not enough servers running (`S.active[i]` additionally satisfied). Since the transitions for both guard labels do not provide any distribution type (both are passive), we model an immediate synchronous transition. If there is at least one server running a guards statement (line 31) allows to synchronously process a job and dequeue the next job. Here the `<process>` transitions in the `Service` behavior provide the exponential distribution type, which is also the composed synchronous distribution, since the `Q.<deq>` transitions are passive. The third guards statement (line 32) is responsible for the deactivation of servers, if the queue is empty. Finally, the Probability statements (lines 34 .. 38) specify the same measures as in Fig. 2.
In order to emphasise the modularity aspects of LARES we propose a third LARES specification for the same model on the LARES website [11]. There, the service process (from Fig. 3) is split into distinct server instances (represented by a Module definition), which yields a non-trivial instance tree with intermediately instantiated Modules. Surely the reachable state space gets enlarged since symmetries in the service process are unfolded. However, these symmetrical states can be aggregated by lumping.

3 Textual Editor Plugin

In order to support the LARES modeller we implemented an editor environment for the LARES DSL (domain specific language) which runs in Eclipse. For this purpose we use the Xtext framework [8] which generates default implementations for both the DSL and the editor components. Xtext needs two models: a DSL grammar model and a DSL object model (see Fig. 4). The *LARES Grammar* model is textually specified within the Xtext editor and conforms to the *Xtext Grammar* meta-model. Here all parser rules for the grammar of the LARES language are specified in an EBNF-like style. A *textual LARES* model (conforming to the LARES Grammar model) is parsed by the XText framework and transformed into an instance of the DSL object model. In our case the object model is provided by the *LaresDsl* model which in turn conforms to the *Ecore* meta-model. Concretely, the LaresDsl model corresponds to the textual notation and provides types which represent the entities of the LARES language.
In order to be able to parse a textual LARES model into the LaresDsl object model, a connection between both is specified inside the LARES Grammar model. As an example consider Fig. 5. Here a parser rule named `Transition-Statement` parses a character string starting with “Transitions from” and transforms to a *TransitionDefinition* object in the object model. The *Transition-*

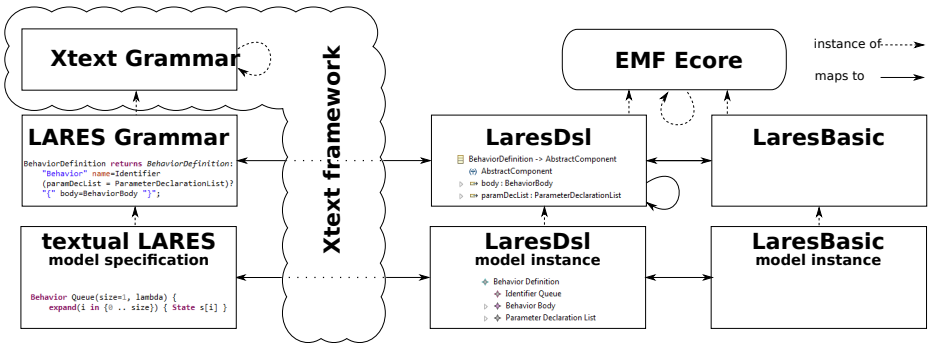


Fig. 4. Interrelations between the LARES models and their model instantiations

Definition object is fed with the following information: a cross-reference to an existing source state of type *State* created when an `ID` is parsed, an optional list of indices for the source state by delegating to the `IndexList` parser rule and a non-empty list of transitions which delegate to the `Transition` parser rule.

```
TransitionStatement returns TransitionDefinition:
   "Transitions" "from"
   sourceState = [State | ID] (sourceStateIndexList = IndexList)?
   (transitions += Transition)+;
```

Fig. 5. Specification of `TransitionStatement` parser rule in the LARES Grammar model. Italic text denotes the connection to the LaresDsl object model.

As mentioned, LaresDsl corresponds to the textual notation of LARES, which also allows to represent "dynamic" LARES constructs such as parameters, arithmetic expressions and expand statements. These dynamic parts of LaresDsl can be made "static" by an in-place transformation to a resolved LaresDsl model (cf. Sec. 3.2). We further define the *LaresBasic* model, which abstracts LaresDsl from these dynamic constructs and textual pecularities such that a graphical editor can be directly supported (cf. Sec. 5). The transformations between LaresDsl and LaresBasic are implemented in a rule-based fashion by applying ETL (Epsilon Transformation Language [18, 2]).

3.1 Editor Features

In addition to the mentioned parsing functionality, Xtext also generates a minimal default implementation for the infrastructure of the LARES editor, which is briefly outlined in the following. We also show how some of these features were modified and new features added in order to be able to specify valid LARES models and assist the modeller while editing. For this reason Fig. 6 shall serve as an example of a parametrised `Behavior` definition named `B`.

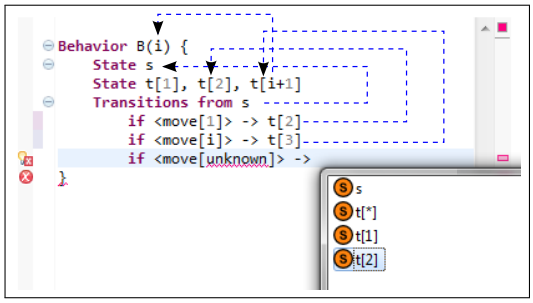


Fig. 6. Behavior definition with cross-references (dashed arrows), error markers and content assist.

Xtext comes with a default *scoping* functionality, which allows to restrict the view on objects which can be cross-referenced. Since LARES follows the object-oriented paradigm, e.g. allows to build models in a modular and hierarchical way, information can be encapsulated. For this reason we modified the default scoping feature by restricting the visibility of objects that can be cross-referenced. As an example, Behavior `B` in Fig. 6 defines four states, which are cross-referenced in the transition statement as source or target states. Note that the `t` states are indexed with arithmetic expressions. The parameter `i` is cross-referenced in the index expression of state `t[i+1]`. All defined parameters and states are only visible inside the Behavior, i.e. they cannot be referenced from the outside.
The cross-references can be visualised to the modeller by the *hyperlinking* feature, which allows to jump in the editor from a textual region representing a cross-reference to the textual element representing the referenced object. In order to establish a cross-reference, Xtext comes with a default *linking* feature. As already mentioned, LARES models are parametrizable and allow to model arithmetic expressions and set expressions. These expressions belong to the LaresDsl object model and are not evaluated on-the-fly while editing. For this reason, we made the linking smarter by matching the index list in patterns in order to create a cross-reference which is either correct or a provisional suggestion to the first matched appropriate object. As an example, the target state `t[3]` references to the suggested state definition `t[i+1]`, since all other state definitions starting with `t` do not match the pattern '3' of the index list. Note that without evaluating the arithmetic expressions, it is not guaranteed to find the semantically correct cross-reference.

Consider for this Fig. 7: The source states `t[1,1]` and `t[i,i]` could match all of the state definitions, since the parameter `i` is not known and the expression `1+1` is not evaluated. Therefore they are referenced to the first found match `t[1,i]`. In the same way the target state `t[2,1]` is referenced to `t[i,1]`. However, for the target state `t[2,2]` no defined state can be

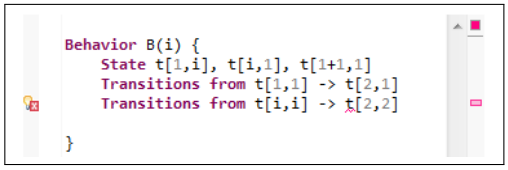


Fig. 7. Linking feature: The cross-reference `t[2,2]` cannot be matched to any state.

surely matched, regardless of which value the parameter `i` takes. Therefore no cross-reference is created and the LARES model is considered as invalid.

The semantics of a LARES model is defined by transformation into the Stochastic Process Algebra CASPA [22] (cf. Fig. 1). In order to be able to perform this transformation, the LARES model has to fulfil several validation constraints. The default *syntax validator* automatically checks whether the textual model conforms to the LARES Grammar model. If it violates the grammar, an error marker is created and visualised at the corresponding location in the editor. Furthermore, an Ecore *reference validator* also checks if all cross-references in the model have been established. Remember that Xtext restricts the view on objects that can be cross-referenced by the scoping feature. As an example, Fig. 6 shows two violations. First, the index `unknown` is marked as erroneous, since there is no defined parameter named `unknown` in the scope of the reference. Second, the target state in a transition is compulsory (by the grammar) and has to be specified. In order to be able to check the LARES model for additional semantical validation constraints, Xtext provides a *validator* feature. We describe in the following only a few of the implemented constraints checked during validation. First of all, names of objects of specific types have to be unique in scope. Second, the ranges for numerical values are restricted, e.g. rates of an exponentially delayed transition have to be positive. Last but not least, definitions which induce cyclic dependencies cannot be evaluated and have to be checked.

As an example, in Fig. 8 the `Condition` statements `b` and `c` are cyclic and same also holds for `d[1]`. In the `expand` statement for each value of `i` in the range between 2 and 10 a `Condition` named `d[i]` is defined, which also induces cycles, if this statement were expanded into nine separate `Condition` statements. However, since arithmetic and set expressions are not evaluated the cyclic dependency can not be assured and thus not checked in the validator feature. We will deal with this problem in more detail in Sec. 3.2.

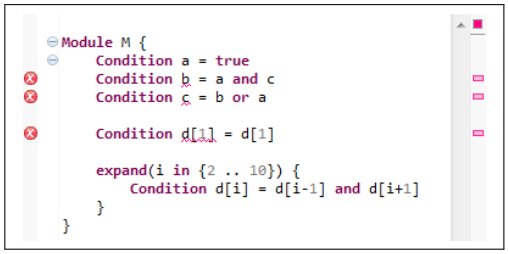


Fig. 8. Validation feature: Elements can be checked for cyclic dependencies.

In order to support the LARES modeller in the user interface, the *content assist* feature provides context-dependent textual proposals. By default these proposals contain all grammar elements and cross-referencable objects which are allowed in the context of the cursor. As an example, Fig. 6 shows proposals for states in the scope of the target state reference. Since LARES statements do not employ separators, many parser rules from different semantical contexts might be applicable, s.t. the plethora of default proposals might rather confuse LARES newcomers. For this reason we modified the content assist by a context-dependent filtering of the default proposals. Moreover, we added some typical templates and com-

ments to the proposals [16].
As already mentioned, the parsing process transforms a textual model to the LaresDsl object model. This transformation can be reverted by the *serializer* feature which plays an important role in Sec. 3.2. Note that the LARES language is descriptive, which means that the order of the statements does not matter. For this reason different textual representations might yield the same LaresDsl object model when parsed. Therefore the serializer transforms into one of the possible textual representations. In order to make the serialised textual representation more readable (following some textual modelling conventions), a *formatter* feature has been implemented, which is responsible for the pretty printing functionality [16].

3.2 Deep Validation

LARES models are analyzed by performing several sequential transformation steps, which all together resolve parameters and references to conditions and labels (forward resp. guard labels), thereby ending up in an intermediate model. This resolved model can then be either transformed to a SPA specification [22] or by performing a reachability analysis into an LTS [15] (cf. Fig. 1) upon which state-based computations are performed in order to return the desired measure values. Each of these transformations can only be performed correctly if some assumptions or necessary restrictions are met in the source model of the corresponding transformation. In order to further assist the modeller, these assumptions can be validated before executing the transformation. If a constraint has proven to be invalid, the modeller can be notified with additional information about the error type and its origin in the source model.
The first transformation step performed on a LARES model is the parameter resolution phase. Here arithmetic expressions and set expressions are evaluated and looping and recursive constructs are expanded (e.g. the `expand` statements). As an example Fig. 9 shows a LARES specification with parameters and an equivalent specification without parameters.

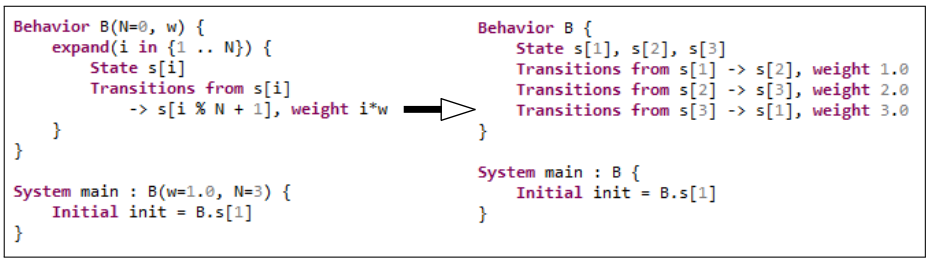


Fig. 9. Parameter resolution: Two equivalent LARES specifications. Left: parametric dependencies and `expand` statement. Right: all parameters are resolved.

We implemented this transformation with EPL (Epsilon Pattern Language [2]) as an in-place transformation, i.e. an arbitrary LaresDsl model is transformed

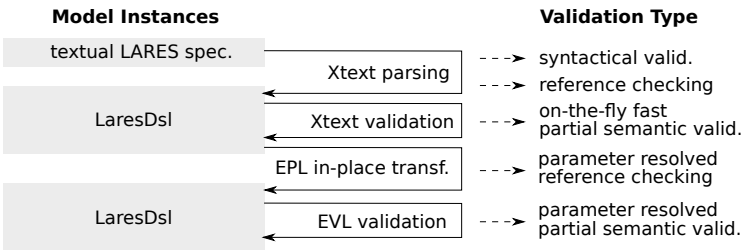


Fig. 10. Validation process

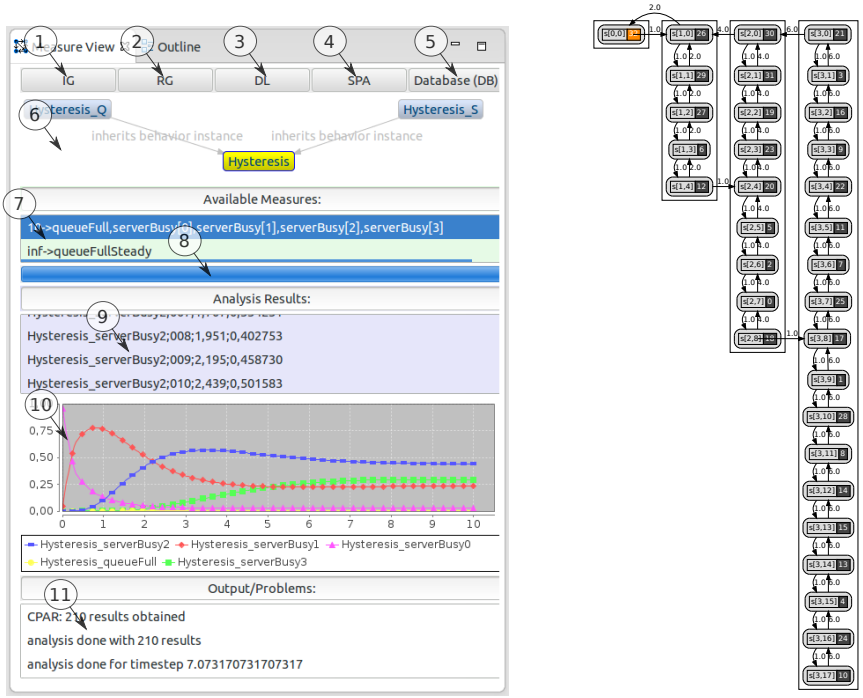
into a resolved LaresDsl model. The EPL transformation is defined in a descriptive pattern-based way, which allows to split the rather complex recursive and iterative transformation into several independent parts, thus improving the maintenance and correctness of the transformation. In Fig. 9 the Behavior `B` is instantiated in the `System main` and defines the values `w` and `N` which can be substituted in the Behavior definition. Now all arithmetic expressions which refer to these parameters can be evaluated to numbers. Therefore the set expression in the `expand` construct can be resolved s.t. in turn all arithmetic expressions which refer to `i` can be resolved and finally evaluated.
If a necessary cross-reference to some object cannot be established in the parameter resolution phase, it is considered as invalid and the modeller is informed with an error marker on the appropriate cross-referencing element in the editor. For instance, if the EPL transformation is performed on the model in Fig. 8 the resolution of the `expand` statement produces such a cross-reference error, since after resolution there is no `Condition` definition named `d[11]`. In order to be able to create these error markers in the editor, we accompany the EPL transformation with a *Trace* model, which maps a resolved LaresDsl element to the element from which it originated by transformation. Once all cross-references are established, the resolved LaresDsl model can be serialised to its textual representation which can be viewed or edited with the LARES editor and manually investigated and validated by the modeller. Finally, the resolved LaresDsl model is further validated by constraints defined in EVL (Epsilon Validation Language [19, 2]), which roughly speaking perform (among others) once more the fast on-the-fly validations from the Xtext validator but now for all resolved LaresDsl elements [16]. The whole validation workflow is summarised in Fig. 10.
Note that the validation process for the analysis presented here is not complete, since further transformations in the LARES workflow are not pre-validated. The reason is that for some model constraints an expensive reachability analysis on the composed state space has to be performed.

4 View Plugin

In order to extend the graphical user interface (GUI), the LARES View Plugin has been developed to carry out the analysis in a comfortable way. It aims to support the experiments by steering the analysis process, the visualisation

of results and their management. It has been developed as an Eclipse View Plugin component, which reacts on specific editor events to synchronise to modifications applied to the model. Whenever the current model has changed, the instance tree is internally reconstructed, then converted to a DOT graph [5] and drawn on an SWT composite [7] using the ZEST library [3]. Fig. 11(a) shows the GUI of the View Plugin: the instance graph of a LARES model is instantaneously visualised on the GUI (cf. ⑥). A more elaborated version (including the provided Condition and forward identifiers of all instances as well as the associated Behavior definitions) can be serialised in the DOT format if required (cf. ①). In order to check whether the intended semantics is met for some specific construct inside a LARES model, if e.g. a modeller is not certain about the transformation semantics, the plugin allows to dump the generated SPA specification (cf. ④). For smaller models, also a reachability analysis to generate a DOT graph of the composed state space might help for model validation (cf. ② and Fig. 11(b)). Moreover, the ability of the CASPA tool to determine the k-most probable paths into a set of target states [24], for instance deadlock states, is accessible using the GUI and can be directly visualised (cf. ③ and ⑥). The Probability measure statements specified in a LARES model are extracted (cf. ⑦): the transient measures are grouped following the associated analysis timepoint, while the steady-state measures are within a single group since their timepoint is implicitly considered infinity. For all transient measures, a dialogue is opened when performing their analysis to ask for additional equidistant intermediate timepoint analysis. Then the transformation workflow as implemented by the LARES library is performed to construct the SPA specification which is accepted and analysed by the CASPA process algebra solver. The obtained results are collected in a list (cf. ⑨) for which a copy&paste feature has been implemented to allow a transfer to external tools such as spreadsheet packages. Simultaneously, the results are visualised (cf. ⑩) in terms of an xy-plot drawn by the JFree chart library [6]. Since each timepoint requires its own independent analysis run, parallel execution following the number of logical CPU cores is supported. There is also a progress monitor for the analysis (cf. ⑧). Finally, output information and errors occurring while parsing, transformation or analysis are reported (cf. ⑪) to give feedback to the user in the case of issues that have not already been detected by the semantic validation features included in the textual Editor Plugin. These outputs can help the user to fix his model or to obtain further knowledge about the intermediate steps of the transformation or analysis of a model. It is planned to connect to a database to save and manage the experiments performed (cf. ⑤). There the results obtained from the solvers are stored together with the version of the specification used for analysis, e.g. to compare different model parametrisations.

Internally, the View Plugin is a mixed Scala/Java project that uses the AKKA actor library [1] for decoupling the internal components via message passing and assuring smooth usability of the GUI, since each component is realised as an actor which itself is a lightweight process. In consequence, the GUI does not block the whole IDE, despite performing an analysis burdening the CPU.



(a) Using the LARES View Plugin GUI (b) Resulting State Space

Fig. 11. Analysis of the multiple parallel hysteresis model from Fig. 2

5 Ongoing Development

Currently a *graphical* LARES editor is also in development, using the Eclipse Graphiti framework [4] and plain Java. Beyond the application of graphical layout algorithms, Graphiti directly supports EMF models. To facilitate the interchange of model information, we specified the LaresBasic model (cf. Sec. 3). Two diagram types, one for a Module definition and another one for the Behavior definition have been developed. The class structure generated from the LaresBasic model is used to deal with a graphical LARES representation. The Graphiti classes have been extended to realise the graphical notation and tooling. Thus standard features such as *delete*, *resize* or wizards to construct diagrams were inherited and could be used out-of-the-box. Furthermore, the layout information of a LARES model had to be separated from the content, and the *drill-down* functionality (i.e. construct/open new diagrams inside another via double-click) needed to be implemented to enable a smooth navigation among the different diagram entities. As a medium term goal, we aim to complete the graphical editor to ease the LARES modelling for new users not yet familiar with the textual syntax. As a long term goal, we aim for a hybrid graphical/textual editor that integrates both approaches, in order to experience the best from both worlds.
As already indicated, it is foreseen for the LARES View Plugin to establish a

database binding to save and manage experiments. For that purpose, a simple database query language will be applied to store current experiments or to gather results of older ones for further processing or comparison.

6 Conclusion/Outlook

We have presented an informative overview on the LARES IDE with a focus on the textual Editor Plugin. This plugin was created in a model-driven way by employing the Xtext and Epsilon frameworks. The editor is enhanced beyond the semantically correct scoping mechanism with several assisting features, like a smart content assist, cross-reference linking and a fast partial model validation facility. These extended editor features support the user by a facilitated access to the modelling world of LARES. Furthermore, it allows to perform a deep validation by transforming parametrised LARES models into parameter-free LARES models. The LARES View Plugin has also been detailed in this paper. This user interface enables a modeller to analyse LARES specifications and calculate the measures of interest. Intermediate representations, such as the reachability graph or the generated process algebra model, can be used to validate the model beyond the syntactical and semantic aspects captured by the grammar and the transformations applied, thus ensuring that the model indeed has the desired behaviour. The graph representations (instance graph or reachability graph) and the xy-plots can easily be serialised in the SVG file format for further use e.g. in publications.
We have also already extended the LARES language with the capability to specify Markov reward models and Markov decision processes [14, 15], such that measures regarding the performability of a LARES model can be specified and an optimal policy w.r.t. to such a measure can be computed. In the future, in addition to the issues discussed in Sec. 5, we plan to provide the necessary tooling for these extensions on the LARES website [11].

Acknowledgments. We thank Dominik Schwindling for his work on the graphical Editor Plugin, and Deutsche Forschungsgemeinschaft (DFG) who supported this work under grants SI 710/7-1 and by DFG/NWO Bilateral Research Programme ROCKS.

References

1. Akka toolkit (2013), `http://akka.io/`
2. Epsilon (2013), `http://www.eclipse.org/epsilon/`
3. Graphical Editing Framework (GEF) (2013), `http://www.eclipse.org/gef/`
4. Graphiti (2013), `http://www.eclipse.org/graphiti/`
5. Graphviz - Graph Visualization Software (2013), `http://www.graphviz.org/`
6. Jfree (2013), `http://www.jfree.org/`
7. Standard widget toolkit (2013), `http://www.eclipse.org/swt/`
8. Xtext (2013), `http://www.eclipse.org/Xtext/`

9. Bozzano, M., Cimatti, A., Roveri, M., Katoen, J.P., Nguyen, V., Noll, T.: Codesign of dependable systems: a component-based modeling language. In: Proc. of the 7th IEEE/ACM int. conf. on Formal Methods and Models for Codesign. pp. 121–130. MEMOCODE '09, IEEE Press, Piscataway, NJ, USA (2009)
10. Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An opensource tool for symbolic model checking. In: Proc. of the 14th Int. Conf. on Computer Aided Verification. pp. 359–364. CAV '02, Springer-Verlag, London, UK (2002)
11. Design of Computer and Communication Systems Group (Inf 3) UniBw: LARES website (2013), `http://lares.w3.rz.unibw-muenchen.de/`
12. Gouberman, A., Riedl, M., Schuster, J., Siegle, M.: A Modelling and Analysis Environment for LARES. In: Schmitt, J.B. (ed.) Measurement, Modelling, and Evaluation of Computing Systems and Dependability and Fault Tolerance, LNCS, vol. 7201, pp. 244–248. Springer Berlin Heidelberg (2012)
13. Gouberman, A., Riedl, M., Schuster, J., Siegle, M., Walter, M.: 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. pp. 153–160. Taylor & Francis Ltd. (2009)
14. Gouberman, A., Riedl, M., Siegle, M.: A Modular and Hierarchical Modelling Approach for Stochastic Control. In: MIC '13: Proc. of the 32nd IASTED Int. Conf. on Modelling, Identification and Control. ACTA Press (2013)
15. Gouberman, A., Riedl, M., Siegle, M.: Transformation of LARES performability models to continuous-time Markov reward models. In: Proc. 7th Int. Workshop on Verification and Evaluation of Computer and Communication Systems (VECOS '13). eWiC, British Computer Society (2013)
16. Grand, C.: Extension of a textual editor for the specification language LARES - Model transformation, validation and feature development. Master's thesis, Bundeswehr University Munich (2013)
17. Hartmanns, A.: MODEST - A unified language for quantitative models. In: FDL. pp. 44–51. IEEE (2012)
18. Kolovos, D.S., Paige, R.F., Polack, F.A.: The Epsilon Transformation Language. In: Proceedings of the 1st international conference on Theory and Practice of Model Transformations. pp. 46–60. ICMT 08, Springer-Verlag (2008)
19. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: On the evolution of OCL for capturing structural constraints in modelling languages. In: Rigorous Methods for Software Construction and Analysis, pp. 204–218. Springer-Verlag (2009)
20. Kühn, P.J., Mashaly, M.: Performance of self-adapting power-saving algorithms for ICT systems. In: IFIP/IEEE International Symposium on Integrated Network Management (IM 2013). pp. 720–723 (2013)
21. Point, G.: AltaRica: Contribution à l'unification des méthodes formelles et de la sûreté de fonctionnement. Thèse de doctorat, Université Bordeaux I (2000)
22. Riedl, M., Schuster, J., Siegle, M.: Recent extensions to the stochastic process algebra tool CASPA. In: Quantitative Evaluation of Systems, 2008. QEST '08. Fifth International Conference on. pp. 113–114 (2008)
23. Riedl, M., Siegle, M.: A LAnguage for REconfigurable dependable Systems: Semantics & Dependability Model Transformation. In: Proc. 6th Int. Workshop on Verification and Evaluation of Computer and Communication Systems (VECOS '12). pp. 78–89. eWiC, British Computer Society (2012)
24. Schuster, J., Siegle, M.: Path-based calculation of MTTFF, MTTFR, and asymptotic unavailability with the stochastic process algebra tool CASPA. Journal of Risk and Reliability 225(4), 399–406 (2012)