Codesign of Dependable Systems: A Component-Based Modeling Language

Marco Bozzano, Alessandro Cimatti, Marco Roveri
Embedded Systems Group
Fondazione Bruno Kessler, Trento, Italy
Email: {bozzano,cimatti,roveri}@fbk.eu

Joost-Pieter Katoen, Viet Yen Nguyen, Thomas Noll
Software Modeling and Verification Group
RWTH Aachen University, Aachen, Germany
Email: {katoen,nguyen,noll}@cs.rwth-aachen.de

Abstract

This paper presents a model-based approach to system-software co-engineering which is focused on aerospace systems but is relevant to a much wider class of dependable systems. We present the main ingredients of the SLIM modeling language and give a precise interpretation of SLIM models by providing a formal semantics using networks of event-data automata. The major distinguishing aspects of this component-based approach are the possibility to describe nominal hardware and software operations, hybrid (and timing) aspects, as well as probabilistic faults and their propagation and recovery. As our approach bears strong resemblance to the standardized AADL (Architecture Analysis and Design Language), a secondary contribution of this paper is a formal semantics of a large fragment of AADL including its Error Model Annex.

1. Introduction

Hardware/software (HW/SW) co-design of safety-critical systems such as on-board systems that appear in the aerospace domain is a very complex and highly challenging task. Component-based design is an important paradigm that is helpful to master this design complexity while, in addition, allowing for reusability. The key principle in component-based design is a clear distinction between component behavior (implementation) and the possible interactions between the individual components (interfacing). Components may be structured in a hierarchical manner akin to an AND-composition in the visual modeling formalism Statecharts [1]. The internal structure of a component implementation is specified by its decomposition into subcomponents, together with their HW/SW bindings and their interaction via connections over ports. Component behavior is described by a textual representation of mode-transition diagrams.

As safety-critical systems are subject to hardware and software faults, the adequate modeling of faults, their likelihood of occurrence, and the way in which a system can recover from faults, are essential to a model-based approach for safety-critical systems. Although several formal approaches to component-based design have been recently reported in the literature [2], [3], [4], [5], error handling and modeling has received scant attention, if at all. Another shortcoming of many proposals is the lack of connection to a notation that is used and understood by design engineers. We attempt to overcome these shortcomings by enriching a component-based modeling approach with appropriate means for modeling probabilistic fault behavior. To warrant acceptance by design engineers in, e.g., aerospace industry, our approach is strongly inspired by the Architecture Analysis and Design Language (AADL; [6], [7]) and its recent Error Model Annex [8] for modeling probabilistic fault behavior.

This paper proposes the System-Level Integrated Modeling (SLIM) language in which engineers are provided with convenient ways to specify, amongst others, nominal hardware, as well as software operations, probabilistic faults and their propagation to subcomponents, error recovery and degraded modes of operation. We provide a gentle introduction to the language by means of a small example, and present in detail a formal semantics of SLIM that provides the interpretation of SLIM specifications in a precise and unambiguous manner. As a semantical model for the nominal system behavior we use networks of event-data automata. Such automata are in fact symbolic means to model (besides the usual automata ingredients) discrete data variables and continuous evolution such as the advance of time and variables whose temporal behavior is described by differential equations.

Error behavior is defined by probabilistic finite-state machines, where error delays may be governed by continuous random variables (in particular negative exponential distributions). The integration of nominal behavior and error models follows an approach advocated in [9], and basically boils down to a product construction of an event-data and a finite probabilistic automaton. As our approach is strongly based on AADL and its Error Model Annex, our semantics can also be considered as a formal interpretation of a large fragment of AADL. To the best of our knowledge, a semantics covering the nominal and error behavior in AADL has not been published yet.

We firmly believe that a rigid mathematical foundation as provided in this paper yields a solid basis for developing

Funded by ESA/ESTEC under Contract No. 21171/07/NL/JD

software tools to support the modeling and analysis of AADL specifications. In fact, we are currently implementing prototypical software tool support for the automated analysis of SLIM specifications, addressing the issues of correctness, safety, and performability. The focus is on model checking of both nominal and the probabilistic error behavior [10]. This toolset is being developed in the context of the European Space Agency (ESA) project COMPASS (Correctness, Modeling, and Performance of Aerospace Systems) [11] and will be applied to several case studies by a major industrial developer of aerospace systems.

This paper is further organized as follows. Section 2 provides a gentle introduction to the SLIM modeling language focusing on nominal behavior, error modeling, and safety aspects. The main technical contribution is provided in Section 3 which provides a detailed account of the semantics of SLIM. Section 4 describes the tool-set that is currently under development, Ssection 5 describes related work, and Section 6 concludes and provides some pointers to current and future research.

2. Syntax

The SLIM language is introduced by means of a simple example, a processor failover system as given by the AADL model depicted in Fig. 1. This system is composed of two processors and a monitor. Only one processor is active at a time. It takes new jobs (via port `new_Job`) and notifies finished jobs (via port `fin_Job`). In case the active processor fails, the monitor switches to the other processor.

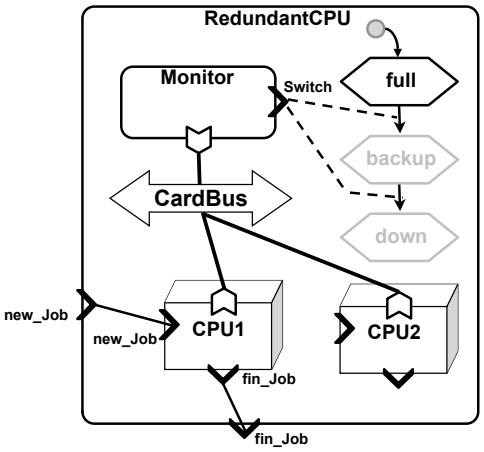

Figure 1. AADL model of the processor failover system while no processor has failed, which is mode `full`.

This small case already highlights the component-based nature of systems. In SLIM, this is reflected by having software (indicated by the keywords **`process`**, **`thread`** or **`thread group`**) and hardware (keywords **`processor`**, **`memory`**, **`bus`**, and **`device`**) components as first-class language constructs. Sets of interacting components can be grouped into a composite component which is indicated by the keyword **`system`**. This facility enables the modeler to manage the system's complexity through introducing component *hierarchy*. The top-level component thus represents the system's global behavior whereas its subcomponents describe specific and concrete behavior of system parts. The resulting hierarchical system model, also referred to as *nominal model*, describes the system behavior under normal operation. This is complemented by an *error model* which expresses how the system can fail, i.e., it models how faults may affect normal operation and may lead to a degraded mode of operation. A part of nominal components may be designated as components that deal with recovering from errors; they are referred to as FDIR (Fault Detection, Identification, and Recovery) components.

2.1. Nominal Behavior

SLIM distinguishes component types and their implementation. A *type* describes an interface for interacting with other components. In the SLIM snippet below, the `RedundantCPU` takes as inputs `new_Job` events and outputs `fin_job` events. Events are transmitted between components in a synchronous manner, i.e., the sending and receiving of an event occurs in a "handshaking " operation without involving any buffering. Here multiway communication is supported, that is, an outgoing event can be received by several components.

Additionally, it is possible to define incoming and outgoing *data* ports (which are absent in the snippet below), which are typed. As opposed to event ports, the information exchange via data ports is continuous, i.e., an update operation on an outgoing data port immediately affects the connected incoming data port(s) without requiring any explicit receiving operation.

```
system RedundantCPU
features
  new_Job: in event port;
  fin_Job: out event port;
end RedundantCPU;

system Monitor
features
  Switch: out event port;
end Monitor;

processor CPU
features
  new_Job: in event port;
  fin_Job: out event port;
end CPU;
```

```
bus Bus
end Bus;
```

Whereas the component type specifies a component's interface, the component *implementation* describe how and when these interactions are performed. It provides the internal structure consisting of **subcomponents** that outlines component hierarchy, **connections** for connecting event ports, and **modes** and **transitions** that describe the component's behavior. The following SLIM fragment provides an implementation of the RedundantCPU.

```
system implementation RedundantCPU.Impl
subcomponents
  CPU1: processor CPU.Impl
    accesses CardBus in modes (full);
  CPU2: processor CPU.Impl
    accesses CardBus in modes (degraded);
  Mon: system Monitor.Impl
    accesses CardBus;
  CardBus: bus Bus.Impl;
connections
  event port new_Job -> CPU1.new_Job
    in modes (full);
  event port CPU1.fin_Job -> fin_Job
    in modes (full);
  event port new_Job -> CPU2.new_Job
    in modes (degraded);
  event port CPU2.fin_Job -> fin_Job
    in modes (degraded);
modes
  full: activation mode;
  degraded: mode;
  down: mode;
transitions
  full -[Mon.Switch]-> degraded;
  degraded -[Mon.Switch]-> down;
end RedundantCPU.Impl;

processor implementation CPU.Impl
modes
  idle: initial mode;
  processing: mode;
transitions
  idle -[new_Job]-> processing;
  processing -[fin_Job]-> idle;
end CPU.Impl;

bus implementation Bus.Impl
end Bus.Impl;
```

Port connections either occur between a component and its subcomponents or between subcomponents. The latter models interaction between components at the same hierarchical level whereas the former deals with inter-level interaction. Port connections can be *mode-dependent*. That is to say, the way in which components interact can depend on their modes. A mode change may thus imply an amendment of the interconnection structure between components. In the above example, the port connections relay inputs and outputs from component RedundantCPU to the active processor, which is by default CPU1, in mode full. In mode degraded, however, the inputs and outputs are relayed to CPU2. In contrast to relaying input and output events, port events can also be "absorbed" (like new_Job) and spontaneously generated (like fin_Job).

The mode transition system —basically a finite-state automaton— describes how the component evolves from mode to mode while performing events. Clock invariants (not in the above example) constrain the residence time in a mode and flow-expressions specify how continuous variables evolve while residing in a mode. This is similar to timed and hybrid automata [12]. A mode transition is of the form m_1 -[e **when** g **then** a]-> m_2. It asserts that the component evolves from mode m_1 to mode m_2 on the occurrence of event e (the trigger event) provided the guard g, a Boolean expression that may depend on the component's (discrete and continuous) variables, holds. On transiting, the assignment a which may change a data subcomponent (like the Boolean Failure below) or outgoing data port, is applied. The presence of event e, guard **when** g and assignment **then** a is optional. Mode transitions may give rise to modifying a component's configuration: subcomponents can become (de-)activated and port connections can be (de-)established. This depends on the **in modes** clause, which can be declared along with port connections and subcomponents.

```
system implementation Monitor.Impl
subcomponents
  Failure: data bool initially false;
modes
  Stay: initial mode;
  Change: mode;
transitions
  Stay -[when Failure]-> Change;
  Change -[Switch]-> Stay;
end Monitor.Impl;
```

In the SLIM model detailed so far, the Boolean Failure is never changed, and thus stays false. Consequently, the Switch event never occurs and component RedundantCPU will never switch from CPU1 to CPU2. A CPU failure has to occur in order to make the Monitor switch meaningful. This is modelled by an error model.

2.2. Error Behavior

The error model describes how faults affect the nominal system behavior. Similarly to nominal models, error models

consist of a type and associated implementation. An error model type declares *error propagations*, which are similar to event ports, but in contrast are linked implicitly. Error propagation "connections" are based on whether components access a common bus, or whether they are in a super-subcomponent relation. This implicit linking reflects the oblivious and pervasive nature of error models. Besides error propagations like `CPUfails`, error states are defined, such as `Activation` below.

```
error model MonError
features
  OK: activation state;
  Failed: error state;
  CPUfails: in error propagation;
end MonError;

error model CPUError
  features
    CPUfails: out error propagation;
    OK: initial state;
    Activation: error state;
    Propagated: error state;
end CPUError;
```

The error model implementation describes when errors can occur and how these affect the state of the error model. This behavior is described by a *probabilistic* finite-state automaton [13], [14]. Transitions occur due to spontaneous **events** which may be annotated with a rate that indicates the expected number of occurrences per time unit (event `fault` below, e.g., occurs on average 40 times per time unit). This information is leveraged for performing probabilistic analyses. Transitions can also occur because of error propagations. Inwards error propagations cause a transition (like `CPUfails` in `MonErr.Impl`), outwards error propagation are generated by following the transition (like `CPUfails` in `CPUError.Impl`).

```
error model implementation MonError.Impl
  events
    tau: error event;
  transitions
    OK -[CPUfails]-> Failed;
    Failed -[tau]-> OK;
end MonError.Impl;

error model implementation CPUError.Impl
  events
    fault: error event
      occurrence poisson 40;
  transitions
    OK -[fault]-> Activation;
    Activation -[CPUfails]-> Propagated;
end CPUError.Impl;
```

As error models bear no relation with nominal models, an error model does not influence the nominal model unless they are linked through *fault injection*. A fault injection consists of three parts: a state s in the error model, a data subcomponent d in the nominal model, and the fault effect given as the expression e. Semantically, the error and nominal model are concurrently active. On entering error state s, the assignment $d := e$ is carried out, i.e., the data subcomponent is assigned with the fault effect. For the redundant CPU example, the nominal model `Monitor.Impl` and the error model `MonError.Impl` are linked through the injection of `Failure := true` upon entering error state `Failed`. Multiple fault injections between error models and nominal models are possible. The combined model, i.e., the integration of the nominal models, error models, and the fault injections, is called the *extended model*.

3. Semantics

To enable the trustworthy modeling and analysis of systems, our SLIM language is equipped with a formal semantics that provides the interpretation of SLIM specifications in a precise and unambiguous manner. The meaning of a nominal specification is basically defined on two levels, distinguishing between the local behavior of an active component and the interaction between active components via ports and connections. This interaction is highly dynamic as local mode transitions can cause subcomponents to become (in-)active, and can change the topology of event and data port connections. On the level of the formal semantics this means that both the activation status of components and their interconnection relation depend on the modes of the individual components.

3.1. Local Component Behavior

The local behavior of a component is described by a finite automaton model which supports both discrete (i.e., state-based) and continuous (i.e., hybrid and timed) aspects. Formally, an *event-data automaton (EDA)* is a tuple of the form

$$\mathfrak{A} = (M, m_0, X, v_0, \iota, E, \rightarrow)$$

where

- M is a finite set of *modes*,
- $m_0 \in M$ denotes the *initial mode*,
- X is a finite set of *variables*, partitioned into *input variables*, IX, *output variables*, OX, and *local variables*, LX,
- $v_0 \in V$ is the *initial valuation* where V denotes the set of all *valuations*, that is, mappings that assign values to the elements of X,
- $\iota : M \rightarrow (V \rightarrow \mathbb{B})$ specifies the *mode invariants* (where we assume that $\iota(m_0)(v_0) = \mathsf{true}$),

- E is a finite set of *events*, partitioned into *input events*, IE, and *output events*, OE, and
- $\rightarrow \subseteq M \times E_\tau \times (V \rightarrow \mathbb{B}) \times (V \rightarrow V) \times M$ is a finite *(mode) transition relation* where $E_\tau := E \cup \{\tau\}$. Transitions are represented in the form $m \xrightarrow{e,g,f} m'$, and e, g, and f are called the *trigger*, the *guard*, and the *effect*, respectively. Here an empty guard or effect is interpreted as `true` and the identity, respectively.

The operational interpretation of transition $m \xrightarrow{e,g,f} m'$ is that the EDA can switch from mode m to m' on the occurrence of event e provided the guard g holds (where g may depend on the variables in X). On taking the transition, local and output variables are updated according to the function f. An EDA may stay in mode m under valuation v as long as the mode invariant $\iota(m)(v)$ holds.

The association of an EDA with a given SLIM component specification is straightforward:

- The meaning of modes in the SLIM component and in the EDA is identical.
- Incoming and outgoing data ports are interpreted as input and output variables, respectively, and data subcomponents are interpreted as local variables.
- Events in the EDA are either SLIM event ports, or are used to represent the event communication between a supercomponent and one of its (active) subcomponents. In the second case, they are of the form $c.e$ where c is the identifier of the subcomponent, and e its event port. Here an incoming event port in the subcomponent gives rise to an output event in the EDA of the supercomponent, and vice versa.

For example, the SLIM specification of the `RedundantCPU` component as presented in the previous section yields the EDA $\mathfrak{A} = (M, m_0, X, v_0, \iota, E, \rightarrow)$ with

- $M = \{\texttt{full}, \texttt{degraded}, \texttt{down}\}$, $m_0 = \texttt{full}$,
- $X = \emptyset$ and $v_0 = []$, i.e., the empty function,
- $\iota(\texttt{full}) = \iota(\texttt{degraded}) = \iota(\texttt{down}) = \mathsf{true}$,
- $IE = \{\texttt{new\_Job}, \texttt{Mon.Switch}\}$, $OE = \{\texttt{fin\_Job}\}$,
- $\rightarrow = \{\texttt{full} \xrightarrow{\texttt{Mon.Switch}} \texttt{degraded}, \texttt{degraded} \xrightarrow{\texttt{Mon.Switch}} \texttt{down}\}$.

The operational semantics of an EDA is given as a labeled transition system whose states, called *configurations*, are pairs of modes and valuations. Transitions either model the passage of time, involving an update of the non-discrete variables, or are internally triggered by events, including the invisible event τ. The second case requires the guard of the respective transition to be enabled, and then modifies the valuation of the variables according to the transition effect.

Formally, the *semantics* of an EDA is given by the labeled transition system

$$(Cnf, \kappa_0, L, \rightarrow)$$

with

- the set of *(local) configurations* $Cnf := M \times V$,
- the *initial configuration* $\kappa_0 := (m_0, v_0) \in Cnf$,
- the set of *transition labels* $L := \mathbb{R}_{>0} \cup E_\tau$, and
- the *(local) transition relation* $\rightarrow \subseteq Cnf \times L \times Cnf$, given by
  - *time transition*: $(m, v) \xrightarrow{t} (m, v + t)$ if $t \in \mathbb{R}_{>0}$ and the invariant stays valid for t time units, that is, $\iota(m)(v + t') = \mathsf{true}$ for each $0 < t' \leq t$. (Here the notation $v + t$ denotes the update of the clocks and hybrid variables after passing of t time units.)
  - *internal or event transition*: $(m, v) \xrightarrow{e} (m', v')$ if $e \in E_\tau$ and in the current mode m, an e-transition is enabled where the invariant of the target mode is valid after applying the transition effect, that is, there exists $m \xrightarrow{e,g,f} m'$ in $\mathfrak{A}$ such that $g(v) = \mathsf{true}$ and $\iota(m')(v') = \mathsf{true}$ for $v' := f(v)$.

3.2. Global System Behavior

In order to determine the global system behavior we have to specify how the EDAs that represent single components interact with each other. This interaction is highly dynamic as local transitions can cause subcomponents to become (in-) active, and can change the topology of event and data port connections. On the level of the formal semantics, which is given as a network of EDAs, this means that both the activation of the component EDAs and their interconnection depend on the modes of the individual EDAs.

Formally, a *network of event-data automata (NEDA)* is a tuple of the form

$$\mathfrak{N} = ((\mathfrak{A}_i)_{i \in [n]}, \alpha, EC, DC)$$

where $n \geq 1$, $[n] := \{1, \ldots, n\}$, and

- each $\mathfrak{A}_i$ is an EDA $\mathfrak{A}_i = (M_i, m_0^i, X_i, v_0^i, \iota_i, E_i, \rightarrow_i)$,
- $\alpha : M \rightarrow 2^{[n]}$ is the *activation mapping* (where $M := \prod_{i=1}^n M_i$ is the set of *global modes*),
- $EC : M \rightarrow (\{i.oe \mid i \in [n], oe \in OE_i\} \times \{j.ie \mid j \in [n], ie \in IE_j\})$ is the *event connection mapping*, and
- $DC : M \rightarrow (\{i.x \mid i \in [n], x \in OX_i\} \times \{j.y \mid j \in [n], y \in IX_j\})$ is the *data connection mapping*.

The activation mapping α specifies the active (sub)components in a given global mode, i.e., $\alpha(m_1, \ldots, m_n)$ is the set of active components in mode $(m_1, \ldots, m_n)$, where m_i denotes the mode of component i. EC and DC provide the mode-dependent event and data connection, e.g., $EC(m_1, \ldots, m_n)$ denotes which output events are connected to which input events.

Just as a single component is formalized by an EDA, a complete SLIM specification can be represented by a NEDA:

- The activation mapping, α, yields (the indices of) those EDAs that are active in a given global mode. It is directly determined by the SLIM specification: the main component (usually indexed by 1) is always active,

and each subcomponent of an active component that is activated in the current mode of that component is also active. This is identical to the treatment of AND-states in Statecharts [1].
- For a given global mode of the system, all end-to-end (that is, multistep) connections from an outgoing (event or data) to an incoming (event or data, respectively) port of an active component are determined, and are taken into account in the EC and DC mappings. In the first case, also the (implicit) event connections between a supercomponent and its direct subcomponents have to be considered.

For example, the collection of SLIM component specifications from the previous section gives rise to a NEDA with five EDAs (for `RedundantCPU`, `CPU1`, `CPU2`, `Mon`, and `CardBus`) with the following activation mapping:

$$\begin{aligned}\alpha(\texttt{full}, m_1, m_2, m_3, m_4) &= \{1,2,4,5\},\\ \alpha(\texttt{degraded}, m_1, m_2, m_3, m_4) &= \{1,3,4,5\},\\ \alpha(\texttt{down}, m_1, m_2, m_3, m_4) &= \{1,4,5\},\end{aligned}$$

where $m_1, m_2 \in \{\texttt{idle}, \texttt{processing}\}$, $m_3 \in \{\texttt{Stay}, \texttt{Change}\}$, and where we assume that the mode set of the last component just contains an initial mode, that is, $M_5 = \{m_4\}$. Moreover each event connection set contains the entry $(4.\texttt{Switch}, 1.\texttt{Mon.Switch})$.

The *semantics* of a NEDA is given by the labeled transition system

$$(\mathit{Cnf}, \kappa_0, L, \Longrightarrow)$$

which is defined in terms of the local transition systems $(\mathit{Cnf}_i, \kappa_0^i, L_i, \rightarrow_i)$ (for $i \in [n]$) of the constituent EDAs as follows:

- the set of *(global) configurations* is given by $\mathit{Cnf} := \prod_{i=1}^{n} \mathit{Cnf}_i$,
- the *initial configuration* is $\kappa_0 := (\kappa_0^1, \ldots, \kappa_0^n)$,
- the set of *transition labels* is $L := \mathbb{R}_{>0} \cup \{\tau\}$, and
- the *(global) transition relation*, $\Longrightarrow \subseteq \mathit{Cnf} \times L \times \mathit{Cnf}$, is given as follows where $\kappa := (\kappa_1, \ldots, \kappa_n) \in \mathit{Cnf}$ with $\kappa_i = (m_i, v_i)$:
  - *time transition*:

    $$\kappa \overset{t}{\Longrightarrow} (\kappa_1', \ldots, \kappa_n')$$

    if $t \in \mathbb{R}_{>0}$ and all active EDAs are involved in the time step, that is, for each $i \in [n]$, $\kappa_i \xrightarrow{t} \kappa_i'$ if $i \in \alpha(m_1, \ldots, m_n)$, and $\kappa_i' := \kappa_i$ otherwise.
  - *internal transition*:

    $$\kappa \overset{\tau}{\Longrightarrow} \mathit{nxt}(\kappa, \{(i, \kappa_i')\})$$

    if the i^{th} EDA is active and can perform an internal step, that is, there exists $i \in \alpha(m_1, \ldots, m_n)$ and $\kappa_i' \in \mathit{Cnf}_i$ such that $\kappa_i \xrightarrow{\tau}_i \kappa_i'$.
  - *multiway communication transition*:

    $$\kappa \overset{\tau}{\Longrightarrow} \mathit{nxt}(\kappa, \{(j, \kappa_j') \mid j \in J \cup \{i\}\})$$

    if the i^{th} EDA is active and performs an output transition which is forwarded to at least one connected EDA that offers a corresponding input transition, that is, there exists $i \in \alpha(m_1, \ldots, m_n)$, $oe \in \mathit{OE}_i$, and $\kappa_i' \in \mathit{Cnf}_i$ such that $\kappa_i \xrightarrow{oe}_i \kappa_i'$, and

    $$J := \{j \in \alpha(m_1, \ldots, m_n) \setminus \{i\} \mid \exists ie \in \mathit{IE}_j \text{ s.t. } (i.oe, j.ie) \in \mathit{EC}(m_1, \ldots, m_n), \kappa_j \xrightarrow{ie}_j \kappa_j'\} \neq \emptyset.$$

Here the auxiliary function $\mathit{nxt} : \mathit{Cnf} \times 2^{\bigcup_{i \in [n]} \{i\} \times \mathit{Cnf}_i} \rightarrow \mathit{Cnf}$ reflects the impact of mode transitions occurring in the constituent EDAs, taking the current global configuration (first parameter) and the new local configurations (second parameter) into account. This impact is defined as follows:

- Each EDA occurring in the second parameter enters the new configuration.
- If the corresponding mode transition disconnects an input variable (that is, the variable occurs as a target in a data connection of the current mode but in no data connection of the new mode), then that variable is assigned its initial value.
- If this transition activates a (direct) subcomponent that was inactive before, then that subcomponent enters its starting mode, and its data elements obtain their initial values. In the same way, every (direct or indirect) subcomponent of that component is handled, thus restarting the whole subsystem.
- Finally, the global configuration is made consistent by copying the value of the source variable of each connection in DC to its target variable.

An example global transition sequence can be found at the end of the following subsection.

3.3. Integrating Error Behavior

When it comes to integrating faulty system behavior, first the association between nominal and error models needs to be fixed. As explained in Section 2, this is done by assigning error models to nominal component specifications. Moreover one can define failure effects that are attached to error states in order to specify the impact of a fault to the nominal behavior of that component. Every such effect is given by an assignment to data elements that overrides the nominal transition effects in the presence of an error state.

The actual integration of the nominal and the error model, the so-called *(fault) model extension*, works similarly to the procedure described in [9]. It takes the nominal (mode) model and enriches it by the error model specification, thus producing an integrated model which represents both the nominal and the failure behavior. Informally, this model is obtained as follows:

- Its modes are pairs of nominal modes and error model states.

- The set of event ports is obtained by adding the error propagations to the original event ports, in order to represent the exchange of error information via propagations as event communication.
- Correspondingly, the set of event port connections has to be extended by propagation port connections according to the rules that govern the exchange of error information.
- In the mode transition relation of the integrated model, all possible interleavings between the nominal and the error model have to be considered, taking failure effects into account.

Since an error model definition allows to attach probabilistic information to state transitions (cf. the specification of `CPUError` in the previous section), we have to extend our semantic model correspondingly. Concretely this means that we have to add a new kind of transitions, *Markovian transitions*, to both EDAs and NEDAs. As these transitions are triggered by error events, which are internal to the respective error model, and do not involve any guards or assignments, they can be represented in the form $s \xrightarrow{\lambda} s'$ where s and s' are error states and $\lambda \in \mathbb{R}_{>0}$ is the rate parameter. Thus the transition relation of the integrated model becomes a subset of $(M \times S) \times E_\tau \times (V \to \mathbb{B}) \times (V \to V) \times (M \times S) \cup (M \times S) \times \mathbb{R}_{>0} \times (M \times S)$ where M and S refer to the nominal modes and error states, respectively. Intuitively, $s \xrightarrow{\lambda} s'$ means that after on average $1/\lambda$ time units, s is left and the next state is s'. In case $s \xrightarrow{\lambda} s'$ and $s \xrightarrow{\mu} s''$, state s is left on average every $1/(\lambda+\mu)$ time units, and its direct successor is s' with probability $\lambda/(\lambda+\mu)$ or s'' with the complementary likelihood.

For example, integrating the nominal `CPU` specification with the `CPUError` error model yields the following transitions over $\{\texttt{idle}, \texttt{processing}\} \times \{\texttt{OK}, \texttt{Failed}\}$. Here m and s respectively refer to the current mode and error state of the component.

$$\begin{array}{ccc}
\begin{pmatrix} m:\texttt{idle} \\ s:\texttt{OK} \end{pmatrix} & \underset{\texttt{fin_Job}}{\overset{\texttt{new_Job}}{\longleftrightarrow}} & \begin{pmatrix} m:\texttt{processing} \\ s:\texttt{OK} \end{pmatrix} \\
\downarrow 40.0 & & \downarrow 40.0 \\
\begin{pmatrix} m:\texttt{idle} \\ s:\texttt{Activation} \end{pmatrix} & \underset{\texttt{fin_Job}}{\overset{\texttt{new_Job}}{\longleftrightarrow}} & \begin{pmatrix} m:\texttt{processing} \\ s:\texttt{Activation} \end{pmatrix} \\
\downarrow \texttt{CPUfails} & & \downarrow \texttt{CPUfails} \\
\begin{pmatrix} m:\texttt{idle} \\ s:\texttt{Propagated} \end{pmatrix} & \underset{\texttt{fin_Job}}{\overset{\texttt{new_Job}}{\longleftrightarrow}} & \begin{pmatrix} m:\texttt{processing} \\ s:\texttt{Propagated} \end{pmatrix}
\end{array}$$

Clearly, Markovian transitions also have to be considered in the transition system of the resulting NEDA by adding another subrelation of the type $\mathit{Cnf} \times \mathbb{R}_{>0} \times \mathit{Cnf}$ to the global transition relation $\Longrightarrow$, which yields a so-called *Interactive Markov Chain* [13].

This is illustrated by the following example. As the previous section defines error models for both the `CPU` and the `Monitor` components, we have to extend three of the five EDAs by taking error behavior into account. The following transition sequence illustrates a computation of the corresponding NEDA. Here again the order (`RedundantCPU`, `CPU1`, `CPU2`, `Mon`) (ignoring `CardBus`) is assumed, and the modes of active components are underlined.

$$\left(\begin{array}{c|c|c|c} m\colon \underline{\texttt{full}} & \begin{array}{l} m\colon \underline{\texttt{processing}} \\ s\colon \texttt{OK} \end{array} & \begin{array}{l} m\colon \texttt{idle} \\ s\colon \texttt{OK} \end{array} & \begin{array}{l} m\colon \underline{\texttt{Stay}} \\ s\colon \texttt{OK} \\ \texttt{Failure}\colon \mathsf{false} \end{array} \end{array}\right)$$

$\Downarrow 40.0$ (`fault` event in `CPUError`)

$$\left(\begin{array}{c|c|c|c} m\colon \underline{\texttt{full}} & \begin{array}{l} m\colon \underline{\texttt{processing}} \\ s\colon \texttt{Activation} \end{array} & \begin{array}{l} m\colon \texttt{idle} \\ s\colon \texttt{OK} \end{array} & \begin{array}{l} m\colon \underline{\texttt{Stay}} \\ s\colon \texttt{OK} \\ \texttt{Failure}\colon \mathsf{false} \end{array} \end{array}\right)$$

$\Downarrow \tau$ (`CPUfails` error propagation)

$$\left(\begin{array}{c|c|c|c} m\colon \underline{\texttt{full}} & \begin{array}{l} m\colon \underline{\texttt{processing}} \\ s\colon \texttt{Propagated} \end{array} & \begin{array}{l} m\colon \texttt{idle} \\ s\colon \texttt{OK} \end{array} & \begin{array}{l} m\colon \underline{\texttt{Stay}} \\ s\colon \texttt{Failed} \\ \texttt{Failure}\colon \mathsf{true} \end{array} \end{array}\right)$$

$\Downarrow \tau$ (`Failure` transition in `Monitor`)

$$\left(\begin{array}{c|c|c|c} m\colon \underline{\texttt{full}} & \begin{array}{l} m\colon \underline{\texttt{processing}} \\ s\colon \texttt{Propagated} \end{array} & \begin{array}{l} m\colon \texttt{idle} \\ s\colon \texttt{OK} \end{array} & \begin{array}{l} m\colon \underline{\texttt{Change}} \\ s\colon \texttt{Failed} \\ \texttt{Failure}\colon \mathsf{true} \end{array} \end{array}\right)$$

$\Downarrow \tau$ (`Switch` event)

$$\left(\begin{array}{c|c|c|c} m\colon \underline{\texttt{degraded}} & \begin{array}{l} m\colon \texttt{processing} \\ s\colon \texttt{Propagated} \end{array} & \begin{array}{l} m\colon \underline{\texttt{idle}} \\ s\colon \texttt{OK} \end{array} & \begin{array}{l} m\colon \underline{\texttt{Stay}} \\ s\colon \texttt{Failed} \\ \texttt{Failure}\colon \mathsf{true} \end{array} \end{array}\right)$$

4. Toolset for Formal Analyses

We plan to leverage several existing mature verification tools in a coherent toolset to enable formal analyses of SLIM models. The prototypical architectural outline of the toolset is depicted in Figure 2.

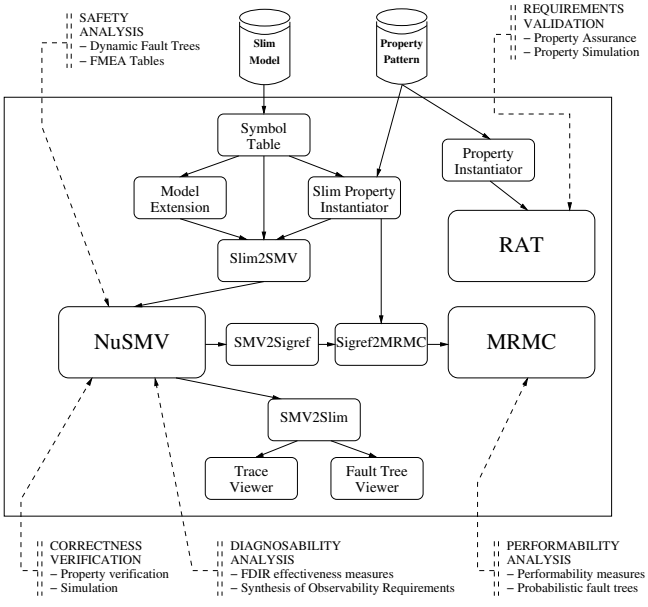

Figure 2. Architecture of the COMPASS toolset.

The core engines of the envisioned toolset are NuSMV, MRMC and RAT (three bigger blocks in the figure above). NuSMV [15] is a model checker for verifying functional

behavior. It employs symbolic techniques that from practice are known to cope well with large transition systems. MRMC [16] is a probabilistic model checker that uses numerical and simulation methods over Markov chains to compute performability characteristics. RAT [17] can be used to determine whether the set of properties that compose the specification is consistent. The design engineer shall not interact with these tools directly as it requires extensive expertise in formal logics and low-level modeling formalisms. Instead, fully automatic translators (the smaller blocks in Figure 2) from SLIM to (and between) the lower-level input formats shall be developed such that operational details of the core engines can be nicely abstracted away from the design engineer. The SLIM semantics, as described in the previous section, are used to preserve semantic equivalence between the translations.

A prototype is working as of April 2009, although it is still under development. Therefore, we shall only sketch the possible analyses supported by the toolkit. More details can also be found in [18].

Simulation techniques can be used to generate a trace that describes a single interaction scheme between components of a nominal (or extended) model. *Model checking* techniques [10] can be employed to exhaustively verify a model against a property. *Probabilistic model checking* techniques [19] can be used to compute the probability of a probabilistic property. Both the non-probabilistic and probabilistic properties can be entered in a user-friendly manner via property patterns; a collection of often-used templates with accompanying high-level descriptions [20], [21]. Translators shall transform these templates to the underlying formal logics, like Linear Temporal Logic and Continuous Stochastic Logic.

To facilitate the analysis of safety and dependability aspects, *fault trees* can be generated via FSAP [9], an extension of NuSMV. Fault trees describe which faults could have caused the system to enter a particular mode, which is called the *top-level event*. If enough probabilistic information is given in the SLIM model, it is also possible to compute the probability of the top-level event using MRMC by transforming the fault tree to its underlying Markov chain [22]. *Failure Mode and Effects Analysis* (FMEA) tables can also be automatically generated using FSAP as well. They describe which modes may be entered when a fault occurs. For this reason, SLIM distinguishes error states (like state `Failed` in `MonErr.Impl`) from nominal states (like state `OK`) in the error model.

For complex systems that implement *Fault Detection, Identification, and Recovery* (FDIR) strategies, it is also important to verify their effectiveness. We plan to leverage the above mentioned analyses for that. Fault detection and recovery can be reduced to a model checking problem. Their probabilistic counterparts can likewise be reduced to a probabilistic model checking problem. Fault isolation, the system's ability to distinguish a fault from other faults, is a case of fault tree analysis. Conducting these FDIR effectiveness analyses requires modeling partial observability. This is realized by introducing an additional attribute, namely **`observable`**, which can be attached to Boolean output data ports. A value change of an observable port may be considered as a fault detection means for a previously occurred fault. The **`observable`** attributes can also be used for diagnosability analysis, which concerns whether a component, regardless of its FDIR system, has proper, or overly, outgoing Boolean data ports from which a faulty state can be deduced. This requires the construction of a TwinPlant model and the accompanying diagnosis condition and then have both fed into NuSMV [23].

5. Related Work

5.1. High-Level Specification Languages

In the first phase of our project, the available high-level formalisms for specifying aerospace on-board systems were identified and evaluated. It was found that the Architecture Analysis and Design Language (AADL; [6], [7]) was the most promising candidate as it targets the specification and analysis of safety-critical real-time distributed computer systems. Developed by a Society of Automotive Engineers (SAE) sponsored committee, it was approved and published as a SAE Standard 2004, and is mainly used in avionics, aerospace, and automotive applications. AADL provides both a graphical and a textual notation for specifying hierarchical system architectures by the interfaces of components, their combinations in subsystems, and their interactions. Moreover it supports the extension of its basic expressiveness by means of so-called annexes, among which the Error Model Annex [8] was particularly interesting for our application.

Another candidate to be mentioned is the Unified Modeling Language (UML; [24]). Although initially coping with software models, UML has been extended so as to model application structure, behavior, and architecture, and also business process and data structures. Unfortunately, the targeted universality of this language implies that the core UML elements do not cope out-of-the-box with specific domains. For example, one of the most important arguments against UML is its loose formalism concerning nominal and error behavior modeling.

It was therefore decided to choose (a subset of) AADL as the starting point for the design of the SLIM Language, enriched by concepts borrowed from its Error Model Annex. The global goal was to make the language, on the one side, expressive enough for the envisaged application domain and, on the other side, amenable to formal methods for checking correctness properties, safety guarantees, and performance and dependability requirements of specifications. On the one

hand, this was achieved by omitting many of the advanced features of AADL (such as property associations, extensions, refinements, prototypes, flows, event-data ports, in-out event ports, certain component categories, etc.). On the other hand, the following constructs have been added to support a more detailed specification of the operational behavior of components, in particular covering the aspect of hybridity:

- *initialization values* for (incoming and outgoing) data ports and data subcomponents,
- the mode type **`activation`**, to support *mode history* (i.e., to allow a re-activated component to resume its operation in the mode in which it was deactivated before),
- explicit *binding relations* between subcomponents (**`stored in`**, **`running on`**, **`accesses`**),
- mode *invariants*, introducing clock constraints and flow equations,
- mode *transition guards*, to enable or disable transitions depending on the values of data elements, and
- mode *transition effects*, introducing value assignments to data elements and clock resets.

5.2. The Underlying Semantic Model

In order to make the high-level system specifications amenable to formal reasoning, a corresponding low-level formalism had to be developed as sketched in Section 3. It shares some similarities with existing approaches for describing component-based systems such as interaction systems [25] and constraint automata [26], as well as formalisms for composing automata such as interface automata [2] and hybrid I/O automata [27]. However, none of these provides the expressiveness that is required to cover all aspects of our SLIM language:

- dynamic reconfiguration of components and port connections,
- explicit representation of data elements with hybrid aspects,
- multiway communication (that is, the transmission of events or data from one sender to several receivers),
- integration of probabilistic error behavior.

5.3. Other Related Work

Another approach to defining the operational semantics of (a subset of) AADL by translating it into a component framework called BIP is recently described in [28]. The incorporation of probabilistic information into a hierarchical modeling formalism has been carried out for the Statemate semantics of Statecharts [29], and has been provided for UML Statecharts using different semantic models such as stochastic Petri nets and probabilistic automata. A probabilistic extension of the component-based modeling formalism Reo [3] has been reported in [30]. The application of probabilistic model checking to FMEA tables in [31] is closely related to the probabilistic analysis of error behavior as is aimed at with the SLIM language. An object-oriented approach towards dependable codesign recently appeared in [32]. Finally, we mention the architectural framework Arcade [33], which can be, roughly speaking, viewed as modeling the Error Model Annex of AADL.

6. Conclusion

In this paper we described the syntax and formal semantics of a component-based modeling formalism that is aimed at the codesign of dependable safety-critical systems. The SLIM language is strongly based on AADL and its accompanying Error Model Annex. Our semantics can thus also be considered as a (first) formal interpretation of AADL and its error annex. We currently bring our results into the AADL standardization body and will in the near future be able to report on practical experiences with using SLIM and its model-checking tools to case studies from the aerospace domain.

Acknowledgment

We thank Benedikt Brütsch, Roberto Cavada, Christian Dehnert, Friedrich Gretz, and Andrei Tchaltsev for their contributions in resolving technical issues in the SLIM semantics.

References

[1] D. Harel, "Statecharts: a visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, 1987.

[2] L. de Alfaro and T. A. Henzinger, "Interface automata," in *9th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering (FSE)*. ACM Press, 2001, pp. 109–120.

[3] F. Arbab, "Reo: A channel-based coordination model for component composition," *Mathematical Structures in Computer Science*, vol. 14, no. 3, pp. 329–366, 2004.

[4] G. Gößler and J. Sifakis, "Composition for component-based modeling," *Science of Computer Programming*, vol. 55, no. 1-2, pp. 161–183, 2005.

[5] S. Moschoyiannis and M. Shields, "Component-based design: towards guided composition," in *Application of Concurrency in System Design (ACSD)*. IEEE CS Press, 2003, pp. 122–131.

[6] "Architecture Analysis and Design Language (AADL)," International Society of Automotive Engineers, SAE Standard AS5506, 2004.

[7] "Architecture Analysis and Design Language (AADL) V2," International Society of Automotive Engineers, SAE Draft Standard AS5506 V2, Mar. 2008.

[8] "Architecture Analysis and Design Language Annex (AADL), Volume 1, Annex E: Error Model Annex," International Society of Automotive Engineers, SAE Standard AS5506/1, June 2006.

[9] M. Bozzano and A. Villafiorita, "The FSAP/NuSMV-SA Safety Analysis Platform," *Int. J. on Software Tools for Technology Transfer (STTT)*, vol. 9, no. 1, pp. 5–24, 2007.

[10] C. Baier and J.-P. Katoen, *Principles of Model Checking*. MIT Press, 2008.

[11] "The COMPASS project web site," http://compass.informatik.rwth-aachen.de/.

[12] T. Henzinger, "The theory of hybrid automata," in *Logic in Computer Science (LICS)*. IEEE CS Press, 1996, pp. 278–292.

[13] H. Hermanns, *Interactive Markov Chains: The Quest for Quantified Quality*, ser. LNCS. Springer, 2002, vol. 2428.

[14] R. Segala and N. Lynch, "Probabilistic simulations for probabilistic processes," *Nordic Journal of Computing*, vol. 2, no. 2, pp. 250–273, 1995.

[15] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "NuSMV : a new symbolic model checker," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 2, no. 4, pp. 410–425, Mar. 2000.

[16] J.-P. Katoen, M. Khattri, and I. Zapreev, "A Markov reward model checker," in *Quantitative Evaluation of Systems (QEST 2005)*. Los Alamitos: IEEE CS, 2005, pp. 243–244.

[17] I. Pill, S. Semprini, R. Cavada, M. Roveri, R. Bloem, and A. Cimatti, "Formal analysis of hardware requirements," in *Design Automation*, 2006, pp. 821–826.

[18] M. Bozzano, A. Cimatti, J.-P. Katoen, V. Y. Nguyen, T. Noll, and M. Roveri1, "The COMPASS approach: Correctness, modelling and performability of aerospace systems," in *28th Int. Conf. on Computer Safety, Reliability and Security (SAFECOMP)*, ser. LNCS. Springer, 2009, to be published.

[19] C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen, "Model-checking algorithms for continuous-time Markov chains," *IEEE TSE*, vol. 29, no. 6, pp. 524–541, 2003.

[20] M. Dwyer, G. Avrunin, and J. Corbett, "Patterns in property specifications for finite-state verification," in *21st Int. Conference on Software Engineering (ICSE)*. IEEE CS Press, 1999, pp. 411–420.

[21] L. Grunske, "Specification patterns for probabilistic quality properties," in *Proc. 30th Int. Conf. on Software Engineering (ICSE '08)*. ACM, 2008, pp. 31–40.

[22] H. Boudali, P. Crouzen, and M. Stoelinga, "Dynamic fault tree analysis using input/output interactive Markov chains," in *IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN)*. IEEE CS Press, 2007, pp. 708–717.

[23] A. Cimatti, C. Pecheur, and R. Cavada, "Formal Verification of Diagnosability via Symbolic Model Checking," in *International Joint Conference on Artificial Intelligence (IJCAI 2003)*. Morgan Kaufmann, 2003, pp. 363–369.

[24] "The Unified Modeling Language," http://www.uml.org/.

[25] G. Gößler, S. Graf, M. E. Majster-Cederbaum, M. Martens, and J. Sifakis, "An approach to modelling and verification of component based systems," in *33rd Conf. on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, ser. LNCS, vol. 4362. Springer, 2007, pp. 295–308.

[26] F. Arbab, C. Baier, J. Rutten, and M. Sirjani, "Modeling component connectors in Reo by constraint automata," *ENTCS*, vol. 97, pp. 25–46, 2004.

[27] N. A. Lynch, R. Segala, and F. Vaandrager, "Hybrid I/O automata revisited," in *4th Int. Workshop on Hybrid Systems: Computation and Control (HSCC 2001)*, ser. LNCS, vol. 2034. Springer, 2001, pp. 403–417.

[28] M. Y. Chkouri, A. Robert, M. Bozga, and J. Sifakis, "Translating AADL into BIP – application to the verification of real-time systems," in *Proc. 1st Int. Workshop on Model Based Architecting and Construction of Embedded Systems*, 2008, pp. 39–53, http://www-verimag.imag.fr/~async/AADL/AADL2BIP.pdf.

[29] E. Böde, M. Herbstritt, H. Hermanns, S. Johr, T. Peikenkamp, R. Pulungan, R. Wimmer, and B. Becker, "Compositional performability evaluation for STATEMATE," in *Quantitative Evaluation of Systems (QEST)*. IEEE CS Press, 2006, pp. 167–178.

[30] C. Baier, "Probabilistic models for Reo connector circuits," *J. Universal Comp. Sc.*, vol. 11, no. 10, pp. 1718–1748, 2005.

[31] L. Grunske, R. Colvin, and K. Winter, "Probabilistic model-checking support for FMEA," in *Quantitative Evaluation of Systems (QEST)*. IEEE CS Press, 2007, pp. 119–128.

[32] B. Theelen, O. Florescu, M. Geilen, J. Huang, P. van der Putten, and J. Voeten, "Software/hardware engineering with the parallel object-oriented specification language," in *ACM/IEEE Int. Conf. on Formal Methods and Models for Co-Design (MEMOCODE)*. IEEE CS Press, 2007, pp. 139–148.

[33] H. Boudali, P. Crouzen, B. Haverkort, M. Kuntz, and M. Stoelinga, "Architectural dependability evaluation with Arcade," in *IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN)*. IEEE CS Press, 2008, pp. 512–521.