FORMALIZING DISTRIBUTED SELF-ADAPTIVE SYSTEMS USING HIGH-LEVEL PETRI NETS

Lorenzo Capra and Matteo Camilli
Dept. of Computer Science, Università degli Studi di Milano,
Via Celoria 18, 20133 Milan, Italy
{capra, camilli}@di.unimi.it

ABSTRACT

Formal models must be adequately supported by software tools in order to aid both the design and development of modern distributed systems. Such systems have a growing complexity due to advanced features, such as adaptive mechanisms, they are equipped with to deal with the dynamical environments in which they operate. In this paper, we introduce a two-layer formal model for self-adaptive distributed discrete-event systems, based on standard (Low- and High-level) Petri nets. This formal framework, supported by a modular software library called PNEMU, can be thought of as a baseline to formalize self-adaptation having a decentralized control. This paper focuses on the description of the model’s operational semantics. Moreover, it outlines the structure and basic usage of PNEMU.

Keywords: High-Level Petri nets, Self-adaptation, Decentralized control, Model checking.

1 INTRODUCTION

Formal models, when adequately supported by software tools, play a crucial role in developing modern-complex distributed systems dealing with different operational conditions due to dynamically changing environments. In particular, guaranteeing important quality attributes, such as correctness, safety, robustness, dependability, in this context is a challenging task. Self-adaptive systems can dynamically and reactively evaluate their own execution context and adjust their behavior accordingly. A popular approach to realize adaptation is using a feedback control loop (de Lemos et al. 2017). The control loop continuously reasons about the current state of the system itself and the surrounding environment to take proper actions. In a distributed (decentralized) setting, different control components may operate concurrently, possibly leading to conflicting/undesired situations. Facing such a complexity both in design and verification phases has been recognized as a major challenge in the field of self-adaptation (Arcaini, Riccobene, and Scandurra 2017). Thus, software tools (based on formal methods) supporting both phases are highly demanded.

In this paper, we present a Petri net-based two-layer formal model for self-adaptive systems and an extensible Python library named PNEMU supporting it (available as open source software at https://github.com/SELab-unimi/pnemu). The model has been informally introduced in (Camilli, Bellettini, and Capra 2018). Here we illustrate its operational semantics in terms of standard High-Level Petri nets (HLPNs) (Jensen and Rozenberg 1991), obtained by composing the two-layers it builds on by means of simple net-operators. The model and the associated software library provide a formal setting for analyzing/simulating adaptable distributed discrete-event systems having a decentralized adaptation control, that leverages the theoretical foundation of HLPNs.
Petri nets (PN) are a sound and expressive formal model for distributed systems. Classical PNs are, however, not adequately equipped (even in their High-level flavors) for representing self-perception and self-adaptation dynamics, typically seen in modern/complex distributed systems. Several attempts to bridge this gap have given rise to PN extensions, most with complex algebraic/functional annotations, where an enhanced modeling capability is very often not adequately supported by convenient analysis techniques and software tools, due to an unclear/unintuitive semantics that limits the applicability of traditional PN analysis techniques, and formal verification activities (such as model checking). Of particular interest are those approaches combining higher-order tokens and the features of object-oriented languages, such as Reference Nets introduced in (Kummer et al. 2004). Even though inspired by the same principles, the formal model implemented by PNEMU relies instead on consolidated PN formalisms. Its operational semantics is defined by an HLPN with first-order tokens, following the ISO/IEC standard (ISO/IEC-JTC-1/SC-7 2004). The focus below is primarily on the compositional way the operational semantics is obtained. We also describe the architecture and the main functionalities of the PNEMU library, in particular, how to define multiple distributed control loops in terms of HLPN components. The interoperability of the library with third-party software tools (in order to carry out formal verification activities) is also briefly discussed. A simple but not trivial example of a self-adaptive system will be used throughout the paper.

The paper is organized as follows. In Sect. 2 we shortly discuss on related works. In Sect. 3 we sketch a few core background concepts making the paper self-contained. In Sect. 4 we introduce a running example used throughout the following sections. In Sect. 5 we present the modeling framework and its operational semantics. In Sect. 6 we describe the main features of its current implementation. In Sect. 7 we briefly discuss on validation/verification capabilities. In Sect. 8 we draw our conclusion and outline ongoing work.

2 RELATED WORK

The approach described in this paper is part of ongoing research activity (Capra and Camilli 2018, Camilli, Capra, and Bellettini 2019) on formal models for distributed self-adaptive systems. Self-adaptation has been widely studied in different research areas, such as software architectures, programming languages, software-engineering, to name a few. We refer to (Weyns et al. 2012, de Lemos et al. 2017) for a comprehensive overview of the landscape of self-adaptive systems. Here, we limit the comparison to formal methods for specifying and verifying distributed self-adaptation, with particular focus to PN-based approaches. In fact, although PNs represent a sound and expressive model of concurrency and distribution, they cannot represent in a natural way structural changes. Several attempts to face this issue gave rise to extensions of PNs with enhanced descriptive expressiveness, but limited support in terms of both analysis techniques and software tools. Most of these extensions, such as those combining “higher-order” tokens and the features of object-oriented programming, do not have a sound semantics, what significantly reduces the applicability of classical analysis techniques. Reference Nets, introduced in (Valk 1998, Cabac et al. 2005), are the representative of this class of formalisms, being supported by a Java software tool called RENew. Another modeling approach, based on object-oriented Petri nets, has been introduced in (Meng 2010) to describe reconfigurable manufacturing systems. This model integrates object-oriented methods, stepwise refinement ideas and Petri nets together. In particular the authors introduce the concepts of macro-place, used to model the aggregation of many processes, and macro-transition, used to link all the related macro-places. This work focuses on the modeling activity. Validation and verification of such a models are currently not supported.

Alternative approaches, rather than extending well-established formalisms with new, complex syntactical features are based on a somehow structured use of them. Meaningful examples are the approaches proposed in (Zhang and Cheng 2006, Camilli, Gargantini, and Scandurra 2018) to model distributed and time-critical self-adaptive systems, respectively. The key idea there is to unfold the nominal and the evolutionary behaviors of a system in a single layer, achieving separation of concerns by means of zones.
The multi-layered modeling approach has been inspired by previous related works using automata-based models introduced in (Merelli, Paoletti, and Tesei 2012). Other models of concurrency have been tailored to represent adaptable and/or evolvable systems. Process Algebras, such as communicating sequential processes (CSP), have been extended in (Bartels and Kleine 2011) to model systems able to react upon external stimuli by changing their internal behavior (e.g., to recover from errors). In this sense, self-healing systems can be viewed as a reactive systems that adapts itself in response to external inputs. Reconfiguration of systems is also supported by the work presented in (Allen, Douence, and Garlan 1998). Systems are described using the Architecture Description Language (ADL) Dynamic Wright. The formal semantics of the ADL is defined by a translation to CSP. Hence, formal properties on the architecture of the system can be verified on the level of CSP. These works focus on architectural aspects of adaptable system design and does not consider functional aspects of the system behavior.

As recognized in (Weyns et al. 2012, Arcaini, Riccobene, and Scandurra 2017) the support to formal specification/verification of self-adaptation with a decentralized control is still low.

3 BACKGROUND

This section provides a brief description of HLPNs and distributed self-adaptive systems having a decentralized adaptation control. A comprehensive theoretical treatment of these topics is outside the scope of this paper. We let the readers refer to the provided bibliography for further details.

3.1 Distributed Self-adaptation.

The term Self-adaptation (Salehie and Tahvildari 2009) is usually employed to characterize systems that can autonomously adapt their behavior at runtime. Self-adaptive systems are able to perceive contextual changes and (re)organize their own features, services, or even structure, in response to these changes. Many models of adaptation assume a single centralized (feedback) control loop that observes the execution context and possibly changes the running system. In essence, a control loop is able to sense the state of the managed subsystem/environment by reading data from a monitor component. Gathered data are analyzed to check whether some adaptation is required. In that case, specific actions are planned, then executed by actuators.

When multiple adaptation concerns (or goals) (de Lemos et al. 2017) have to be considered (e.g., efficiency, reliability, security, etc.), a single centralized loop may not be sufficient to manage the growing complexity (Arcaini, Riccobene, and Scandurra 2017). A way to face it is to concurrently run multiple loops over distributed components. These loops operate and communicate through a shared knowledge in a fully de-
centralized setting, possibly leading to conflict situations. Figure 1 shows this reference model. The two main components of a distributed self-adaptive system are the base layer, containing the managed subsystem (implementing the application logic) and the surrounding environment; and the managing layer, on top of it, implementing the adaptation logic through a number of interacting feedback loops.

3.2 Petri Net Formalisms.

Our modeling framework employs two formalisms, corresponding to the aforementioned layers. The low
(or base) layer is defined by a P/T system (Reisig 1985) enriched with inhibitor arcs, which represents the main functionalities of a system: formally, a 6-tuple \((P, T, I, O, H, m_0)\), where \(P\) and \(T\) are non-empty, finite, disjoint sets holding the places and the transitions of the net. \(I, O, H\) are maps \(P \times T \rightarrow \mathbb{N}\) defining the (input, output, inhibitor) arc connections. \(m_0\) is the initial marking, i.e., a map \(P \rightarrow \mathbb{N}\) representing a system’s state. The firing rule specifies the semantics of a P/T system: a transition \(t \in T\) is enabled in \(m\) iff \(\forall p, I(p, t) \leq m(p) \land (H(p, t) > 0 \Rightarrow H(p, t) > m(p))\). If \(t\) is enabled in \(m\) then it can fire leading to \(m’\), where \(\forall p, m’(p) = m(p) + O(p, t) - I(p, t)\).

The high (or managing) layer, in which self-adaptation aspects are gathered, is defined by interacting High level PN (HLPN) components. We refer to standard HLPNs, as defined in (ISO/IEC-JTC-1/SC-7 2004). In essence, we consider an HLPN as a tuple \((S, T, \ell, M)\), where:

- \(S\) is a finite set of places;
- \(T\) is a finite set of transitions, such that \(P \cap T = \emptyset\);
- \(\ell\) is a labeling function defined as follows:
  - for each \(s \in S\), \(\ell(s)\) represents the type associated with tokens place \(s\) may hold;
  - for each \(t \in T\), \(\ell(t)\) is the guard of \(t\);
  - for each \((x, y) \in (S \times T)\), \(\ell(x, y)\) is the annotation of the (input) arc connecting \(x\) to \(y\), i.e., an expression that evaluates to a multiset of tokens of type \(\ell(x)\);
  - for each \((y, x) \in (T \times S)\), \(\ell(y, x)\) is the annotation of the (output) arc connecting \(y\) to \(x\), i.e., an expression that evaluates to a multiset of tokens of type \(\ell(x)\);
  - \(M_0\) is the net (initial) marking that maps places to multisets of tokens of corresponding types.

The dynamics of an HLPN is given in terms of transition firing modes: a firing mode of \(y\) is a binding \(b\) of variables annotating \(y\)’s incident arcs with corresponding values (ground terms), which satisfies \(y\)’s guard. A firing mode \((y, b)\) is enabled in marking \(M\) if \(\forall (x, y) \in S \times T, \ell(x, y)(b) \leq M(x)\), where \(\ell(x, y)(b)\) is the expression’s evaluation for binding \(b\), and \(\leq\) extends to multisets. The firing of \((y, b)\) withdraws the multiset \(\ell(x, y)(b)\) from every input place \(x\) of \(y\) and adds the multiset \(\ell(y, x)(b)\) to every output place \(x\) of \(y\).

The PNEMU library relies upon SNAKES (Pommereau 2015), a modular Python implementation of standard HLPNs. Intuitively, in SNAKES, tokens are Python objects, transitions are guarded by Python Boolean expressions, and arcs are annotated with Python expressions. SNAKES can be used to either perform step by step simulation or exhaustively explore the state space.

4 A RUNNING EXAMPLE: THE FAULT-TOLERANT MANUFACTURING SYSTEM

In the next sections, we describe both the formal model of self-adaptive systems and the base functionality of the library implementing it with the help of a simple example of fault-tolerant Manufacturing System (MS). The system’s nominal behavior is specified by the P/T system in Figure 2a. Two symmetric production lines (transitions line1, line2) refine raw pieces that are then assembled to get the final product (transition assemble). Raw pieces are loaded from storage into either line, two at once (transition load). Once all of them have been worked, the system restarts (transition reset). A production line is subject to failures, hence
the model includes an abstract specification of a faulty behavior (transition fail). A typical adaptation scenario (taking into account the fault tolerance adaptation concern) involves the reconfiguration of the MS upon a fault (realistically, without shutting the MS down), so that it can continue working with the available product line. The faulty line is detached from the MS, and the behavior of both the loader and the assembler is adapted accordingly: the loader puts two raw pieces at a time on the available line, the assembler machine takes pairs of refined pieces only from that line. Figure 2b shows the reconfigured MS. After the faulty line has been repaired the system is gradually driven back to its nominal configuration.

Despite P/T nets expressiveness, the adaptation procedure (involving changes both in the state and in the structure of the model) cannot be specified in a natural/easy way. In particular, what is far from intuitive, and a potential source of unexpected/undesired situations (such as raw piece loss or, even, deadlocks), is carrying out adaptation without blocking the system execution. Managing adaptation as a separated concern, by means of two different feedback looks, instead, is much easier and safer.

The complete specification of this example can be found along with the additional material provided inside the PNEMU repository.

5 OPERATIONAL SEMANTICS

The rationale of our modeling approach is as follows. The base layer is formalized by a P/T system representing the common dynamics of the managed sub-system (and the surrounding environment). The P/T system is automatically encoded as the initial marking of a “special” HLPN, called emulator (Figure 3), exactly reproducing its behavior.
The emulator’s base types \( P \) and \( T \) represent the base-layer’s places/transitions and should be large enough to cover all possible base layer’s evolutions. Each place of the emulator matches an element of the definition of a P/T system. Place \( \text{mark} \) holds a multiset of type-\( P \) tokens which represents the P/T system’s current marking: the multiset \( N \cdot \text{storage} \), initially held by \( \text{mark} \), corresponds to the MS initial state. The other places encode the underlying P/T net: for instance, the inhibitor arcs of the P/T net in Figure 2a are represented by the multiset of tokens (pairs) below, initially held by place \( \text{inhibitor} \):

\[
1 \cdot \langle \text{broken}, \text{fail} \rangle + 1 \cdot \langle \text{broken}, \text{work1} \rangle.
\]

The guard of transition \( \text{move} \) and the annotations of incident arcs encode the enabling and firing rule for P/T systems, respectively (Camilli, Bellettini, and Capra 2018): there is a one-to-one correspondence between an enabled firing mode of \( \text{move} \) and the corresponding base layer’s transition (bound to \( \text{move} \)’s variable \( t \)) enabled in the P/T marking encoded by place \( \text{mark} \).

By acting on the emulator’s marking is it possible to introspect and, if needed, change the current state/structure of the emulated system. This way, the managing layer can monitor-analyze the execution of the managed subsystem, then plan-execute changes accordingly, in order to achieve specific adaptation goals.

Any interaction with the emulator is safely carried out by high-level subnets using \text{read}/\text{write} primitives. These components, that formalize the feedback loops of the self-adaptive system, are properly connected to the emulator through a simple and sound net-operator. Every loop-net has a \text{begin} and a \text{end} place, with empty pre- and post-sets, respectively.

A small portion of the feedback loop managing a fault occurrence in a MS line is depicted in Figure 5a. The transition with label “\text{lib.getTokens}(p)\rightarrow n” denotes a call to a \text{read} library primitive having an analogous signature. The match between the caller and callee is explained next. This read primitive is used to sample the status of the faulty line, upon which (i.e., when place \( \text{broken} \) is marked) the managed subsystem’s adaptation takes place. Transitions having labels \text{rmOArc} and \text{addOArc} (lines 10-11) are instead two calls to \text{write} primitives (i.e., the \text{actuators}). The former is used to temporarily detach the faulty line (by removing the arc from \( \text{load} \) to \( \text{line1} \)). The latter makes the pair of loaded pieces enter the available line (by increasing the weight of the input arc from \( \text{load} \) to \( \text{line2} \)). Then, the loading of pieces is temporarily suspended (by means of \text{addHArc} primitive), pending pieces on the faulty line are moved to the available line (by means of \text{setTokens}), and so on until the configuration shown in Figure 2b is eventually reached. It should be recalled that all the changes operated by a feedback loop take place concurrently with base layer emulation.

The PNEMU’s run-time support transparently handles the interaction between the base layer and feedback loops so that the resulting model has a clear and sound operational semantics, in terms of a standard HLPN. The base and the managing layers are composed through a \text{transition superposition} (with a possible matching of surrounding arc annotations) that involves, on one side, primitive function calls (in feedback loops) and primitive function definitions, and, on the other side, the emulator’s \text{move} transition and a couple of transitions prefixing each feedback-loop net.

### 5.1 Composing the emulator and the feedback-loops

The emulator net is causally connected to the managing layer as illustrated in Figure 4. The emulator’s \text{move} transition is split into a pair of transitions with mutually exclusive guards, representing observable and non-observable base layer’s events (\text{obsv}(t) is a predicate defined on transition type \( T \) to distinguish observable events). Only the former may trigger the managing-layer. This distinction is realistic, may reduce the model’s complexity (both in structural and state-space terms), and is not restrictive at all. In the MS model the only observable transitions are \text{fail} and \text{repair} (the latter is not present in the system’s initial configuration).
Each feedback-loop net $\text{loop}_i$ is prefixed by a pair of mutually exclusive transitions $\{\text{start}_{\text{L}_i}, \text{skip}_{\text{L}_i}\}$, and suffixed by transition $\text{end}_{\text{L}_i}$. These transitions are connected to a (boolean) semaphore place $\text{semph}_{\text{L}_i}$, (initially marked True), that regulates the access to the control-loop in accordance with the Balking pattern, i.e., new triggers are ignored while the loop is being executed. This prevents the model from becoming unbounded due to the accumulation of trigger-tokens in $\text{loop}_i$ places. Moreover, it avoids base layer's emulation from being affected by possible deadlocks/livelocks caused by badly designed feedback loops. These auxiliary net elements are automatically attached to the user-defined loop-nets.

The interaction between the emulator and the set of $n$ loop-nets is formally defined in terms of a Cartesian product, $\{\text{moveObsv}\} \times \{\text{start}_{\text{L}_1}, \text{skip}_{\text{L}_1}\} \times \cdots \times \{\text{start}_{\text{L}_n}, \text{skip}_{\text{L}_n}\}$, which represents the set of (mutually exclusive) transitions corresponding to all the possible ways of superposing $\text{moveObsv}$ with either $\text{start}_{\text{L}_i}$ or $\text{skip}_{\text{L}_i}$, for all $i: 1...n$. Transition superposition (or merge) is a $k$-ary net operation accomplished: 1) by automatically renaming homonym variables surrounding involved transitions, with the only exception of symbols $\{t, i, o, h, m\}$, that are unified with the corresponding emulator’s variables (their types must coincide, otherwise the whole operation fails); 2) by associating the resulting transition with a guard which is the logical OR of the operands’ guards. The factors of the product are erased at the end of the whole operation.

This interaction model is general and sound, and also permits parametric feedback-loops (acting on any base-layers) to be specified. But it usually requires some pre-processing by loops to evaluate whether they must be responsive to, or simply ignore, an event, depending on whether some conditions (checked on the base layer’s current configuration) are met.

In a number of cases, it may be more convenient to bind a loop to specific events (transitions) of the base layer. This may be done by associating $\text{loop}_i$’s prefixes $\text{start}_{\text{L}_i}, \text{skip}_{\text{L}_i}$ with a guard $g_i$ taking the form $\bigwedge_{j \in O_i} t = t_j$, where $O_i$ is a set of observable events associated with the loop. In order for these guards not to be blocking, an isolated extra transition with complementary guard $\neg(\bigvee_{i:1...n} g_i)$ is added to the managing layer’s interface, so that the Cartesian product formalizing the composition of the base- and the managing layers has an extra term coming from the superposition $\{\text{moveObsv}\} \times \{\text{extra}\}$, matching observable events currently not managed by any loop.

In the fault-tolerant MS example, we might (in an alternative to the current schema) bind one loop to the fault event, the other to the repair event.

### 5.2 Using Library Primitives

Base layer’s introspection/manipulation is carried out through a basic, yet complete, set of primitives implementing read, add, removal, operations on any structural or state elements of the emulated P/T system. These primitives are defined as high-level transitions interconnected with the emulator’s places and may be
invoked by corresponding calls. Matching a call with its definition is denoted instantiation and is performed, once again, by means of transition superposition.

Figure 5b shows the definition, and an example of instantiation, of getTokens read library function. This primitive is used to sample the status of the faulty line, upon which (i.e., when place broken is marked) the managed subsystem starts self-adaptation. It is called from the loop-net by a transition with label “lib.getTokens(p)→n”, whose firing makes the current marking of base layer’s place p (the argument) be assigned to the named return value n (a free variable).

The definition contains the signature and the net elements of the primitive: getTokens reads the content of base layer’s place mark (shared with the emulator net) into variable m, and returns the multiplicity (i.e., the marking) m(p) of the P/T place that is bound to the formal parameter (free variable) p. Each call to a primitive in a loop-net is superposed with the corresponding definition, by carrying out a sort of term matching in their signatures: when superposing the getTokens call with its definition, variable p (in the definition) is matched by the ground term “broken”, whereas variable n (of the loop’s transition) is matched by the term m(p).

In this example, the arguments of a primitive call are supplied “in place”, through a constant. In the alternative they could be indirectly supplied by means of variables annotating input arcs of the caller transition.

6 IMPLEMENTATION

The formal framework introduced above has been implemented as an extensible Python library called PNEMU. The implementation is primarily tailored to model distributed self-adaptive discrete-event systems by means of a clear separation of concerns. The user defines in a modular fashion a HLPN model whose architecture follows the reference two-layer model. Both the base and the managing layers are specified by means of an easy-to-use API.
The library, implemented on the top of SNAKES, has the following main modules:

- **pnemu.base** contains the Emulator component and the structures needed to define a P/T net.
- **pnemu.manager** contains the components needed to model/execute the managing layer. In particular, the FeedbackLoop class allows control loops to be defined in terms of HLPNs. The class AdaptiveNetBuilder is used instead to connect the control loops to the emulator in order to build the overall adaptable system model.
- **pnemu.primitives** provides a number of elementary HLPNs abstracting the notions of sensors and actuators used by control loops. PNEMU comes along with a number of pre-defined LibEntry instances making up a basic, yet complete, set of sensors/actuators. The latter operate atomically, ensuring a consistent encoding of the base-layer.

### 6.1 The Base Module

This module allows the model’s base layer (e.g., the P/T system in Figure 2a) to be defined. This can be done by instantiating class PT, either by loading an existing PNML file or by editing step-by-step the base-layer net. Class PT also provides the means to carry out the interactive simulation, state-space exploration, and visualization of the base layer. Once the base layer has been defined, we need to initialize the emulator component, which reifies the base layer into the high layer. An instance of class Emulator implements the HLPN shown in Figure 3, whose marking encodes the base layer. Figure 6 (line 1) shows the usage of PNEMU to perform these preliminary steps, i.e., base-layer definition and subsequent emulator initialization.

### 6.2 The Manager Module

The components of this module are used to specify (arbitrarily complex) feedback loops in terms of HLPNs. A loop is incrementally built by defining the structure and the annotations of an HLPN, with intuitive method-calls (Figure 6, lines 2-11).

In our example, the specification of the managing subsystem contains another loop, taking into account the load balance concern: this loop allows the MS nominal behavior to be restored after the faulty line has been repaired, so that the raw pieces can be evenly distributed among the available production lines. Once the feedback loops have been created, the overall adaptive system model is made up using AdaptiveNetBuilder (lines 13-16). This module plugs each feedback loop to the emulator net by linking them as explained in Section 5.1. This way, whenever (due to some observable event) the base layer enters a new state (i.e., transition moveObsv fires), the loops are triggered. The build method call (line 16) returns a SNAKES HLPN (i.e., a PetriNet object) that can be in turn inspected, visualized, executed, or analyzed, as discussed in Sect. 7.

### 6.3 The Primitives Module

This module contains a basic, yet complete, set of primitives that constitute an easy-to-use API through which to carry out base-layer introspection/manipulation. These primitives are implemented as high-level transitions wrapped into instances of LibEntry class. An instantiation takes place at any primitive call issued by AdaptiveNetBuilder module, during the construction of the HLPN representing the whole system. Each transition representing a primitive call in a loop-net is superposed with the corresponding definition, by carrying out a simple term matching in their signatures, as explained in Section 5.2.
emulator = Emulator(PT(name='ms', pnml='my/path/ms.pnml'))
loop = FeedbackLoop(name='fault_tolerance')
loop.add_place(name='begin')
loop.add_place(name='sample')
read_primitive = 'lib.getTokens("broken")\rightarrow m'
loop.add_transition(name=read_primitive)
loop.add_input_arc(src='init', dst=read_primitive, Variable('b'))
loop.add_output_arc(src=read_primitive, dst='sample', Variable('n'))
loop.add_transition(name='lib.addHArc("broken","load",1)')
loop.add_transition(name='lib.rmOArc("line1","load",1)')
loop.add_transition(name='lib.addOArc("line2","load",1)')
net = AdaptiveNetBuilder(emulator)
net.add_loop(control_loop=loop, initial_places=['init'])
net.add_loop(control_loop=loop2, initial_places=['init2'])
net.build()
mode = net.transition('move').modes()[1]
# mode = Substitution(m=Multiset(["storage"] * 2), t="fail", ...)
net.transition('move').fire(mode)
assert net.get_marking()('init') == Multiset(["fail"] * 1)

Figure 6: An example of the PNEMU API usage. This code snippet can be executed as a script or in an interactive Python shell.

the list of all the pre-defined read/write primitives provided by PNEMU can be found inside the documentation provided inside the PNEMU public repository. This basic set can be easily extended by adding new user-defined LibEntry objects. The class LibEntry allows new, primitive or even complex, library functions to be created by defining their structure and inscribing it with arbitrary (standard or user-defined) Python functions.

7 FORMAL VERIFICATION

The PNEMU library implements a general and consolidated two-layer modeling framework for self-adaptive systems, based on the use of multiple, distributed feedback loops. It directly supports inspection (of both the state and the structure) of both layers, interactive simulation and visualization of model components. In particular, at each step of the token game it is possible to check for enabled transitions, firing modes, simulate firing, and perform on-the-fly changes to the model’s state/structure. An interactive simulation example is shown in Figure 6 (lines 17-20): line 17 shows the firing modes for the transition move, among which there is the one representing base level’s transition fail; line 19 makes move fire with this mode; line 20 checks that after the firing of base layer’s transition fail, the fault tolerance control loop is triggered.

The modeling approach supported by PNEMU is based on well-established formalisms (i.e., P/T systems and HLPNs following the ISO/IEC standard). This makes it possible the use of consolidated analysis techniques and/or existing software tools. As an example, since the module pnemu.manager produces SNAKES HLPNs (snakes.nets.PetriNet objects) we can exploit the NECO (Fronc and Pommereau 2013) model-checker (designed to work along with SNAKES) to verify the correctness of models with respect to requirements expressed in Linear Temporal Logic (LTL). Thus, it is possible to check: reachability, safety, and liveness properties; local (i.e., relating the base layer only) and/or global (considering both layers)
invariants; adaptation integrity constraints (de Lemos, Garlan, Ghezzi, and Giese 2017). As simple examples, on the running example we checked deadlock freedom: $\Box(\neg \text{deadlock})$, and the following liveness property:

$$\Box([\text{broken}] < \text{marking(mark)} \rightarrow \Diamond(\neg ([\text{broken}] < \text{marking(mark)})))$$

meaning that upon any failures, the MS eventually repairs itself.

8 CONCLUSION

We introduced a formal framework and its supporting software library for modeling adaptable distributed systems. This approach adopts a consolidated two-layer architecture allowing for a clear separation of concerns between the managed and the managing subsystems. In addition to modeling activity, the library has been designed to interoperate with other third-party state space builders/model checkers to perform formal verification. Our implementation has been released as open-source software to encourage usage and replication of experiments. We plan to enhance it with the ability to verify structural properties by means of library primitives that compute structural relations (e.g., conflict, causal connection, mutual exclusion) on P/T elements. Finally, our current primary aim is to try out the whole framework on industrial case-studies.

REFERENCES


Capra, and Camilli


AUTHOR BIOGRAPHIES

LORENZO CAPRA is an assistant professor at the Computer Science dept. of the University of Milan, Italy. He received the Ph.D. degree in Computer Science from the University of Turin (Italy) in 2001. He is especially interested in structural analysis techniques and tools for High-level Petri nets based on algebraic approaches, and in modeling dynamic discrete-event systems.

MATTEO CAMILLI is a postdoctoral research fellow at the Computer Science dept. of the University of Milan, Italy. He received the Ph.D. degree in Computer Science from the University of Milan in 2015. His current research activity focuses on Formal Methods and Software Engineering. He is especially interested in: uncertainty quantification in software development; model-based testing techniques; methods and tools to improve dependability of adaptable and evolvable time-dependent applications.