Fundamenta Informaticae 136 (2015) 1–30
DOI 10.3233/FI-2015-1200
IOS Press

Generalizing the Compositions of Petri Nets Modules

Alexis Marechal[*†]
Centro de Investigaciones en Nuevas Tecnologías Informáticas
Universidad Privada Boliviana, La Paz, Bolivia
alexismarechal@lp.upb.edu

Didier Buchs
Centre Universitaire d'Informatique, University of Geneva
7 route de Drize, 1227 Carouge, Switzerland
didier.buchs@unige.ch

Abstract. Modularity is a mandatory principle to apply Petri nets to real world-sized systems. Modular extensions of Petri nets allow to create complex models by combining smaller entities. They facilitate the modeling and verification of large systems by applying a divide and conquer approach and promoting reuse. Modularity includes a wide range of notions such as encapsulation, hierarchy and instantiation. Over the years, Petri nets have been extended to include these mechanisms in many different ways. The heterogeneity of such extensions and their definitions makes it difficult to reason about their common features at a general level. We propose in this article an approach to standardize the semantics of modular Petri nets formalisms, with the objective of gathering even the most complex modular features from the literature. This is achieved with a new Petri nets formalism, called the LLAMAS Language for Advanced Modular Algebraic Nets (LLAMAS). We focus principally on the composition mechanism of LLAMAS, while introducing the rest of the language with an example. The composition mechanism is introduced both informally and with formal definitions. Our approach has two positive outcomes. First, the definition of new formalisms is facilitated, by providing common ground for the definition of their semantics. Second, it is possible to reason at a general level on the most advanced verification techniques, such as the recent advances in the domain of decision diagrams.

Keywords: System design and verification, Modularity, High-Level Petri nets, Standardization, Composition Mechanism, LLAMAS

[*]This work was partially supported by the COMEDIA project funded by the Hasler foundation, project #2107.
Also works: Centre Universitaire d'Informatique, University of Geneva
[†]Address for correspondence: Centro de Investigaciones en Nuevas Tecnologías Informáticas, Universidad Privada Boliviana, La Paz, Bolivia

1. Introduction

Through the years the original version of Petri nets has been extended to integrate a wide range of notions such as token colors, time and probabilities. In the last decade an initiative has risen to tackle the diversity of the many Petri nets variants, to improve the common understanding of the paradigm and to facilitate the integration of Petri net tools. This initiative took the form of the international ISO-IEC 15909 standard [17], which includes Petri Net Markup Language (PNML) [11], a markup language meant to allow the communication between Petri net tools. Currently, the development of the ISO-IEC 15909 standard is focusing on extensions of the original Petri nets [12].

One of these extensions is modularity, that is, the ability to define complex systems by assembling smaller entities. There are many Petri nets formalisms that include the notion of modularity. This article proposes an approach to standardize the formal definitions of these formalisms. We describe this approach in Section 2, where we propose to define a new modular formalism called LLAMAS. In Section 3 we study the state of the art of modular Petri nets formalisms. We then get to the core of the article by introducing the principles of the LLAMAS language. We describe its composition mechanism in detail, first informally in Section 4 through some simple examples, and then formally in Section 5. We describe the rest of the LLAMAS language by means of an example in Section 6. Section 7 shows a case study of our language, by describing a translation from Modular PNML to LLAMAS. We briefly mention in Section 8 some perspectives on how to perform verification in LLAMAS using Decision Diagrams. We conclude and mention ongoing and future work in Section 9.

This article is an extension of [27]. The main addition in this version is a formal definition of the composition mechanism of LLAMAS, given in Section 5. This formalization is quite complex, but we consider that it is necessary to justify the theoretical foundations of LLAMAS.

2. Preliminaries

2.1. Motivation and Approach

There is a wide variety of mechanisms that have been defined to implement the notion of modularity in Petri nets. Two main approaches have been followed by their authors to define the semantics (i.e., the possible executions) of these mechanisms. Many authors define a translation from their modular formalisms to ”flat” (non-modular) Petri nets. In some cases (e.g., [9]) this translation is quite complex and difficult to understand. In some other cases (e.g., [2, 23]), such translation is simply not attempted as it would be too complex or yield Petri nets with infinitely many places or transitions. The semantics of these formalisms is usually defined as compositions of the transition relations of the leaf (non-hierarchical) modules. This low-level operation is usually very complex. The solution we propose follows the path of the ISO-IEC 15909 standard and PNML. The approach of PNML consists in creating a language to serve as a syntactic platform (i.e., a metamodel) for the definition of Petri nets variants. To extend this syntactic standard to semantic considerations, we propose the approach illustrated in Fig. 1. This approach proposes two artifacts. First, a metamodel to grasp the syntactic concepts of modular formalisms (similarly to what is done in PNML). Second, a modular formalism to serve as a semantic platform for existing and new modular formalisms. We described a first proposition for the metamodel in [25]. Here, we focus on the most complex part of the approach: the semantic platform.

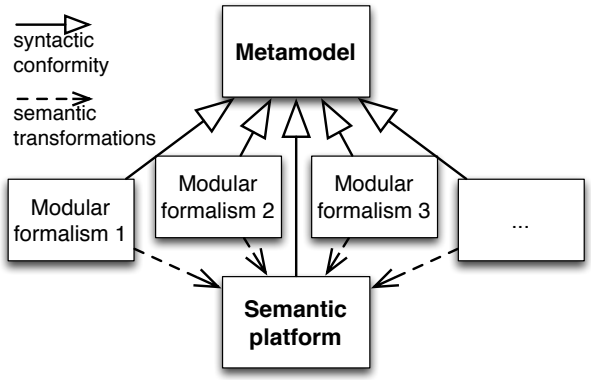

Figure 1: Standardizing the syntax and semantics of modular formalisms

2.2. Formal View of the Approach

Modular formalisms allow to create complex models by composing smaller entities. Let F be a modular formalism, and let M_F be the set of modules that can be expressed with this formalism and, for each module $m \in M_F$, let $Sem_F(m)$ be its Labeled Transition System (LTS). We can define a composite module as an expression $m = \circ_i(m_1, \ldots, m_n)$ where $m_1, \ldots, m_n \in M_F$ are modules and $\circ_i$ is a composition function $\circ_i : \wp(M_F) \rightarrow M_F$. Thus, a module is the composition of modules, that in turn may be compositions of smaller modules, and so on, until reaching the level of leaf modules, i.e., modules that cannot be decomposed anymore. Let us note $LM_F \subseteq M_F$ the set of all the leaf modules in F and $Comp_F = \{o_1, \ldots, o_m\} \subseteq (\wp(M_F) \rightarrow M_F)$ its set of composition operators.

Let us go back to the two approaches we mentioned at the beginning of this section. A syntactic definition of the semantics of F consists in defining the semantics of the leaf modules and a function $Flatten : Comp_F \times \wp(M_F) \rightarrow LM_F$. This function takes a composite module and returns an equivalent leaf module. With it, we can define the semantics of composite modules by defining $Sem_F(\circ_i(m_1, \ldots, m_n)) = Sem_F(Flatten(\circ_i, m_1, \ldots, m_n))$. While this approach is commonly applied to usual fusions of places and transitions, it is not always well suited to more complex cases. Indeed, as we mentioned previously, the $Flatten$ function can be extremely complex or sometimes give undesired results (e.g., infinite nets). Moreover, because the result of this function is a non-modular model, the identity of the different modules is lost at runtime.

On the other hand, a semantic definition is based on an existing formalism SP (for Semantic Platform), that has its own composition mechanisms $Comp_{SP}$ and semantics Sem_{SP}. To create a semantic definition of a formalism F, one must provide two functions, a function $Tr_{mod} : LM_F \rightarrow M_{SP}$ to translate the leaf modules and a function $Tr_{comp} : Comp_F \rightarrow Comp_{SP}$ for the composition operators. Then the function Tr_{mod} is extended to cover composite modules by defining inductively $Tr_{mod}(\circ_i(m_1, \ldots, m_n)) = Tr_{comp}(\circ_i)(Tr_{mod}(m_1), \ldots, Tr_{mod}(m_n))$. Finally, the semantics of F is defined for each model of F as $\forall m \in M_F,\ Sem_F(m) = Sem_{SP}(Tr_{mod}(m))$. A semantic definition has the advantage of preserving the modules identity at runtime. Most formalisms that have semantic definitions use traditional LTS as a semantic platform SP, but they define their own LTS compositions $Comp_{SP}$. Moreover, working at the level of the LTS is a low-level operation, far from the modeling expressivity of Petri nets. This paradigm shift can have negative consequences. For instance, LTS are

not suited for handling concurrency, and often the LTS resulting from the transformation of Petri nets are extremely large.

To avoid this paradigm shift while keeping the benefits of a semantic definition, we propose to use a single Petri nets formalism as a common semantic platform for Petri nets variants. This would facilitate the understanding of their definitions, and would allow reasoning at a general level on computational techniques. This approach is akin to virtual machines in the domain of programming languages. Of course, the semantic platform must be expressive enough to handle at least the existing formalisms. By this we mean that it should be possible to create a translation from each modular formalism to the semantic platform that would preserve its semantics. For any existing modular formalism F with a previously defined semantics $OldSem_F$, we should have:

$$\exists\, Tr : M_F \rightarrow M_{SP} \text{ s.t. } \forall m \in M_F,\ OldSem_F(m) \cong Sem_{SP}(Tr(m))$$

where $\cong$ is an isomorphism between LTS or a state independent equivalence relation such as bisimulation.

2.3. Boundaries of this Article

While an ideal solution would have been to use an existing formalism as a semantic platform, we did not find any expressive enough candidate in the literature. Because of this, we propose a new formalism specifically tailored to serve as a semantic platform for modular Petri nets formalisms. A complete formal definition of the language is out of the scope of this article, but it is given in [24].

Our proposition is strictly limited to the definitions of modularity in the Petri nets formalisms. We rely on the notion of orthogonality as in [12] to limit ourselves to the notion of modularity. By this we mean that we expect other extensions of Petri nets (like time and probabilities) to be compatible with our approach. For instance, we consider that a modular temporal formalism can be defined by combining our modular mechanisms with temporal considerations. Exploring such combinations one of our main research perspectives. In particular, we do not consider the problem of the kind of data types used in the formalism. In the formal definition of LLAMAS in [24] we used Algebraic Abstract Data Types (AADTs) to define the data types, a general specification language that is already used in the ISO-IEC 15909 standard. Here, for simplicity reasons, we use color sets rather than AADTs, and we limit our examples to the use of trivial data types (black tokens, natural numbers).

2.4. State of the Art

Modular PNML [20] is an important candidate for a semantic platform. As a matter of fact, Modular PNML was the initial inspiration for the work presented in this article. Nevertheless, the composition mechanism used by Modular PNML is limited to the syntactical fusion of places and transitions. We consider that this is not enough to express the semantics of some important composition mechanisms defined in the literature, such as the non-deterministic compositions from [6], the parametric compositions from [9] or the complex compositions from [2]. It must be noted that Modular PNML defines a mechanism to compose the definitions of the data types used in the modules. This is an important feature that is not included in LLAMAS because, as mentioned previously, the definition of data types is out of our scope. We consider that our formalism could be easily extended to include Modular PNML's data types definitions, and thus both works can be seen as complementary.

By far, the most widely used modular formalism is the Hierarchical CP-nets [16]. It is a very general formalism, supported by a well established tool, and as such it is an interesting candidate to serve as a standard semantic platform. Nevertheless, similarly to the case of Modular PNML, some composition mechanisms from the literature cannot be represented by this formalism.

Our work can be compared to the process algebras such as CSP [13]. These languages aim to be universal languages that allow to define the behavior of complex formalisms. Nevertheless, the modeling paradigm of process algebras is quite different from the one of Petri nets, and translation between these worlds are often quite complex. LLAMAS avoids this paradigm shift as it is itself a Petri nets formalism.

3. Common Features of Modular Petri Nets Formalisms

To create a standard for modular Petri nets, it is obviously necessary to study the characteristics of the existing formalisms, which we did in a survey [26]. In this section we briefly describe some of the most important characteristics described in that survey.

3.1. The Nature of Modules

3.1.1. Encapsulation

Most modular formalisms include the notion of encapsulation. This means that each module defines a set of public elements that can be accessed by other modules, and a set of internal elements that are hidden and can only be accessed from within the module itself. The set of public elements of a module constitutes its *interface*. In some formalisms, the interface is a set of places and/or transitions of the Petri net itself (sometimes defined by labels on these places and transitions). This is the case of most formalisms that use the classical fusion of places and/or transitions, e.g., [20, 16]. Formalisms that have more complex composition mechanisms usually define interfaces with additional elements that do not belong to the Petri net of the module, e.g., [2, 23].

3.1.2. Hierarchy

The notion of encapsulation is usually linked to the notion of hierarchy. In hierarchical formalisms, a module (called *container*) can encapsulate other modules (called *submodules*). The submodules of a container can in turn be containers of their own submodules, and so on.

3.1.3. Instantiation

The instantiation of modules is a mechanism to define many identical or at least similar modules (instances) based on a blueprint (class). Many formalisms ([20, 16, 9, 2, 23]) define a static instantiation mechanism, which allows to instantiate modules during the definition of the model, prior to any computation. On the other hand, some formalisms define dynamic instantiation, that is, the ability to create instances during the computation of the model. Most of them (e.g., [2, 23, 22]) use Nets Within Nets (NWN) paradigm [32]. One exception is [16] which uses invocation transitions.

3.2. Compositions

The mechanisms that define the interactions between modules are called *composition mechanisms*. The Petri nets literature defines a wide variety of such mechanisms. To describe them, we could make very complex classifications but, for simplicity reasons, we will only mention one classification criterion: state-based vs. event-based mechanisms. Many formalisms define the compositions between modules by relating the places in one module with the places in another module. By far, the most common mechanism for this is the fusion of places. There are some mechanisms that define more complex compositions of places, e.g., the hierarchical transitions in the M-nets family [9]. Other formalisms compose the modules by connecting their transitions. In this case, the composition of a set of transitions produces a new event whose behavior depends on the transitions that were composed. Again, while fusion of transitions is the most common mechanism in this category, there exists more complex mechanisms, like the non-deterministic synchronizations from the CP-nets with channels [6]. Note that many formalisms define both kinds of compositions.

3.3. Summary

We define the structure of a hierarchical module as a tuple $HModule = \langle Spec, Net, I, Subs, Comps \rangle$, where $Spec$ is a data types definition, Net is a Petri net that defines the internal behavior of the module, I is the interface of the module, $Subs$ is a set of submodules and $Comps$ is a set of compositions that define the interactions between these submodules. A complete model may be represented as a tuple $HModel = \langle HModules, Comps, Inst \rangle$, where $HModules$ is a set of modules, $Comps$ a set of compositions between these modules and $Inst$ is an instantiation mechanism.

The elements Net and $Spec$ are out of the scope of this article. The first one, a standard definition of Petri nets, is already covered by the first part of the ISO-IEC 15909 standard. The second one, a standard for modular data types specifications, is precisely defined as one of the main aspects of the formalism presented in [20]. As mentioned previously, the modular data types from [20] could be included in our formalism to include data types modularity.

Instead, LLAMAS focuses on the other elements, the interface I (see Section 6.3), the hierarchical mechanism $Subs$ (Section 6.4), the instantiation mechanism $Inst$ (Section 6.4 again) and the composition mechanism $Comps$, which is the subject of the next section.

4. The LLAMAS Composition Mechanism

In this section we will introduce informally the composition mechanism of LLAMAS. The next section will give complete formal description of this mechanism.

Our goal is to define a single composition mechanism expressive enough to encompass most if not all the mechanisms that have been defined in the literature. We named it the LLAMAS Composition Mechanism (LCM). We mentionned in Section 3 that the composition mechanisms can be classified as state-based or event-based. One possibility was to define the LCM as a mixture of both categories, but we consider that this would be unnecessarily complex. Instead, we defined the LCM as an event-based mechanism, and we ensured that it is expressive enough to *simulate* state-based mechanisms (as explained in Section 2.2). A major advantage of defining the LCM solely as an event-based mechanism lies in the verification activity as it will be mentioned in Section 8.

In general terms, a composition in the LCM is a finite set of *events* that, combined, form another event. The result of a composition may itself participate inside other compositions, forming a hierarchical and recursive structure. Let us start by the most simple example of composition, the fusion of two transitions t1 and t2. This fusion creates a new event c1 that behaves as both transitions fired simultaneously (we note c1 = merge(t1,t2)). This composition c1 may be in turn be fused with a third transition t3, forming a new event called c2 which behaves as the three transitions t1, t2 and t3 fired simultaneously (c2 = merge(c1,t3) = merge(merge(t1,t2),t3)). At this point, one may wonder if it is possible for a transition to participate in more than one composition. In our example, the question is if it would be possible to define two compositions c1 = merge(t1, t2) and c3 = merge(t1, t3), both using independently the transition t1. The answer to this question gives place to the first novel feature of the LCM: an explicit distinction between *bindings* and *calls*.

4.1. Bindings and calls

In the Petri nets literature, there are three possible approaches when considering if the events can participate independently in more than one composition:

- Each event can participate in at most one composition. This is the case of the usual fusion sets, and also of Modular PNML [20].
- Each event may participate in any number of compositions. This is the case of the CP-nets with channels [6], the modular Petri nets from [7] and the family of the M-nets [9].
- Finally, we have a hybrid approach, exemplified by CO-OPN/2 [2] and the reference nets [22]. The events are synchronized by means of *calls*. If an event m1 calls another event m2, it means that m1 cannot be executed independently of this call, but m2 could be called by other events without involving m1.

In the latter case, we say that m1 was *bound* by the composition, and m2 was *called* by the composition (i.e., m1 cannot participate in other compositions but m2 is free to do so). In the LCM, we note this composition merge(bind(m1),call(m2)). Thus, as seen previously, the fusions of transitions in Modular PNML use only *bindings*, and the compositions in CP-nets with channels and the M-nets use only *calls*. We may refine now our previous definition of composition to include the bindings and calls. For now, we will say that a composition in the LCM is defined by two finite sets of events, a collection of *bound* events and a collection of *called* events. For instance, if we define a composition c4 = merge(bind(t1),bind(t2),call(t3)), t1 and t2 cannot participate in any other composition, while t3 can. Binding multiple times the same event (e.g., merge(bind(t1),bind(t1)) is also forbidden.

The distinction between bindings and calls allows to define the semantics of some uncommon and apparently heterogeneous mechanisms from the literature -e.g., the usual fusion sets, the non-deterministic CP-nets with channels and the complex calls from CO-OPN/2- in a single semantic platform. The fact that this distinction is a purely syntactical feature that can be checked statically shows that these apparently very different composition mechanisms are all related to each other at the semantic level. Note that the notion of bindings is more general than the mechanisms found in the literature. Indeed, to the best of our knowledge, no composition mechanism allows to define a behavior like the one of the composition c4 defined above, where two transitions are bound and one is called. Usually, all the transitions are bound, or all of them are called, or only one of them is bound and all the others are called. Graphically, we

represent the compositions by diamonds, bindings by double arrows, and calls by dashed arrows. Fig. 2 shows a graphical representation of c4. It also shows a composition noted c5 = merge(bind(c4),call(t3)). Firing c5 is the same as firing simultaneously t1 and t2 once, and t3 twice.

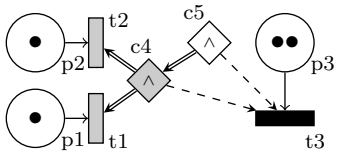

Figure 2: Two compositions.

The shades of the transitions and compositions in Fig. 2 represent their activity, which is the subject of the next paragraph. The symbol ∧ inside the compositions will be explained afterwards. In the following, to simplify the notation, we may omit the keyword call. For instance, c1 = merge(t1, t2) is the same as c1 = merge(call(t1), call (t2)). Bindings will always be explicitly stated.

4.2. Active vs Passive Events

We have seen that some events in the LCM are defined as aggregates of other events. In this context, it would make sense to define some basic events solely for the purpose of being part of a composition, and not with the intent of being executed independently. For instance, the port transitions in the Hierarchical CP-nets [16] are only executed if they are called by some external event. We call such transitions *passive*. The same feature is defined under different names and notations in many modular Petri nets extensions, and as such it is a must have in any comprehensive standard. In most formalisms passive events are only allowed in the interface of the modules, but we generalize this to allow any event (transition or composition) to be active or passive. An event is *active* if it can be executed by itself, and passive if it must be triggered by some composition to be activated. The LTS of a model is labelled only with active events. Graphically, active transitions are represented by black rectangles and active compositions by white diamonds, while their passive counterparts are grey. In Fig. 2, transition t3 and composition c5 are active, and all the other events are passive. The LTS of the net in Fig. 2 is shown in Fig. 3.

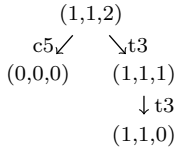

Figure 3: LTS of Fig. 2

4.3. Composition Operators

Up to now we have explored two features of the LCM: the bindings and the active/passive events. We will now explore a more complex feature, the *composition operators*. We said previously that a composition is a set of called events and a set of bound events, and we gave the example c1 = merge(t1, t2). Composition c1 behaves as a classical fusion of two transitions. This is indicated by the use of the composition operator

merge. As we mentioned previously, the fusion is not the only way of combining the behavior of a set of events. There are other possible combinations, and each one will be indicated with a composition operator. Let us refine once again our definition of a composition in the LCM. A composition is now defined as a finite set of bound events and a finite set of called events, all linked by a composition operator. The composition operator determines how the behaviors of the individual events will be combined in order to define a new atomic event. Notice that this is consistent with the mindset of the usual Petri nets, where the atomicity of complex events is already a fundamental principle. In the LCM, we define five composition operators. Three of them are called *behavioral operators* and two of them are called *observers*.

4.3.1. Behavioral Operators

The first behavioral operator is the merge operator, mentioned previously. The other two are the any and the sequence operators. These three operators are taken from CO-OPN/2 [2], a formalism with a powerful compositional mechanism. Let us start with the operator any. A composition c6 = any(t1, t2) corresponds to a non-deterministic choice between t1 and t2. This means that, whenever c6 is executed, one between t1 and t2 is executed, but not both. The merge and any operators are close to the operators from the ITS [29].

The last behavioral operator is the sequence. A composition c7 = sequence(t1, t2) is an event that first executes t1 and then t2. Thus, t2 can use the resources produced by t1. Let us stress out that c7 combines the behavior of t1 and t2 to form a single transactional event. If t2 cannot be fired the whole event is cancelled, and no other event can occur between the executions of t1 and t2. Note that, unlike the merge and any, the sequence operator is not commutative. Consider for instance Fig. 4. It shows three compositions, c8 = any(t1,t2,t3), c9 = any(t4,t5,t6), and c10 = sequence(c8,c9). The any operator in c8 and c9 is represented with a logical disjunction $\vee$, and the sequence operator in c10 with the symbol $>$. Because the sequence operator is not commutative, we must show graphically the order of the elements that are composed. We do this by representing one diamond per composed element, to be read from left to right. There are only two active events in Fig. 4, the transition tx and the composition c10. Whenever c10 is executed, one of the three leftmost places loses its token, and one of the three rightmost places receives one token, all in one action. Both choices are non-deterministic, and thus the P/T Petri net equivalent of c10 would have 9 transitions (not counting tx). Because every execution of c10 is an atomic event, the transition tx will in fact never be executable. Readers familiar with the CP-nets with channels [6] may notice the similarity between that formalism and this example.

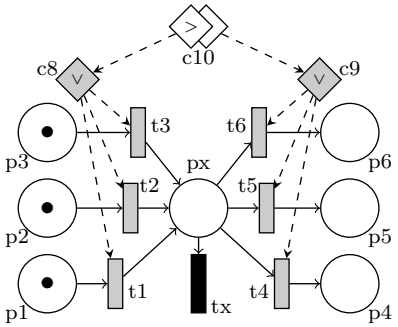

Figure 4: any and sequence operators example.

The three behavioral operators are associative, the merge and any are also commutative and they are both distributive with respect to each other. A distinction between our work and the compositions in CSP [13] is our commitment to transactionality. In the LCM, atomic events are composed to build other atomic events. For instance, composition c10 in Fig. 4 defined a single atomic event. While the parallel composition $\|$ from CSP has a similar behavior, it is not the case for the sequential composition $\gg$. Moreover, our compositions are always defined at the level of single events, whereas in CSP, as in most process algebras, they are defined at the level of the whole CSP processes.

4.3.2. Observers

The behavioral operators allow to create new events by combining the behaviors of other events. While this allows to represent a great number of composition mechanisms, it is not enough for the most complex ones. To cover the most extreme cases, we defined two operators called *observers*. They are the not and the read operators, which are respectively generalizations of the well known inhibitor arcs and read arcs. These operators allow to define special events that do not modify the state of the model, they only observe if some events *can* be fired or not. For instance, a composition not(t1) is enabled if and only if t1 is not. While the not operator checks that an event cannot be executed, the read checks if a given event can be executed. Please note that the observers are a central feature of the LCM. Without them, we could not handle many composition mechanisms such as the stabilization from CO-OPN/2 [2], the complex hierarchy of the M-nets [9], or the special arcs from both the Object Petri nets [23] and the reference nets [22]. Graphically, we represent the not operator with an exclamation mark ! and the read with the symbol $\odot$.

4.4. Parametric Events

Many High Level Petri Nets (HLPNs) formalisms that define event-based compositions allow the events to be parametric. Usually, parametric events allow to transmit information between the composed events (e.g. [6]). In some cases, the parameters are used to choose the recipient of a synchronization (e.g., the communication transitions in [9]). In the LCM, we allow the transitions *and the compositions* to define a list of parameters and a guard. This allows to create compositions like the the one in Fig. 5. We use a dollar sign $ next to the variable names to distinguish them from values. In this figure, there are two parametric transitions, t1($x) which removes a token $x from place p1 and t2($y) which adds a token $y to place p2. $x and $y are the respective formal parameters of the transitions. There is also a composition c11($a) = merge(t1($a), t2($a+1)). In c11, $a and $a+1 are the respective effective parameters of t1 and t2. When c11 is executed with the effective parameter 2, the token with value 2 is removed from p1 and a token with value 3 is added to p2.

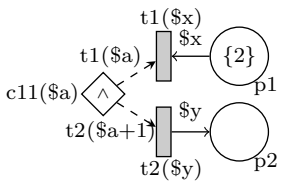

Figure 5: Parametric events.

4.5. Recursion

We saw in the example of Fig. 2 that an event can participate multiple times in the same composition (transition t3 was called twice by composition c5). Similarly, a composition can participate inside itself, i.e., it can be recursive. As is usual in recursive frameworks, there is a danger of defining infinite computations that must be taken care of by the modeler. Fig. 6 is an example of recursion. It shows a transition t1 that removes a token from place p1 and a composition c12 = sequence(t1,any(not(t1),c12)). This composition first calls t1, and then either t1 cannot be executed again (i.e., p1 is empty), or c12 is called again. Thus, during the execution of the composition c12, the place p1 is emptied in a single step. This behavior is represented in Fig. 7, which shows the LTS of the example, labeled by c12, and the detail of the computation of this LTS. With this pattern, we obtain the behavior of the well know reset arcs.

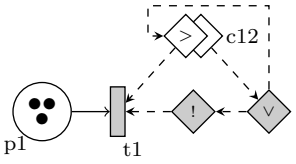

Figure 6: Recursion example.

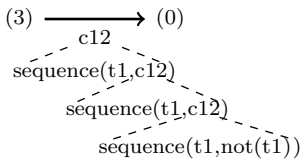

Figure 7: Behavior of composition c12.

4.6. Graphical and Textual Notation - The Case Against Labels

Through this section we have introduced the compositions with both a textual and a graphical notation. The graphical version of our compositions explicitly connects the elements that are composed by means of arrows. Some formalisms note their compositions by means of labels in the composed elements (e.g., [9, 32, 22]). Usually, this notation is difficult to read, as it defeats the intuition behind the graphical notation. On the other hand, a graphical representation of complex compositions may produce bloated graphics that can also be difficult to read, and our formalism is not an exception to this rule. To solve this problem, we propose to use both a graphical and a textual notation to describe our models, inspired by CO-OPN Tools [1]. Note that our current implementation (see Section 8) only uses a textual notation, but we plan to implement a graphical version in the future.

5. The LCM - Formal Semantics

In the original version of this article [27], we introduced LLAMAS informally, without any formal definition. Formal definitions were left out as a technical report, and later included in a PhD thesis [24]. The formal definition of LLAMAS fills 64 pages of that thesis and thus it clearly cannot be included in a standard sized article. Nevertheless, we feel that it is important to have some formal definitions to substantiate the foundations of LLAMAS. Because of that, we will formalize here the LCM, which was presented in Section 4. With this, the central core of the formalism (and the source of its behavioral expressivity) is formalized. Other modular features of LLAMAS (such as encapsulation, hierarchy and dynamic instantiation) are informally introduced by means of a simple example in Section 6.

We also simplified the LCM itself to facilitate its understanding. First, we do not include in this formalization the feature that we called *bindings* in Section 4. Indeed, bindings are a purely syntactical

feature, without any semantic impact, and including them would only pollute the formalization of the semantics. Second, the original version of LLAMAS is defined over the well know Algebraic Petri Nets (APNs), a very expressive variant of HLPN that has been used in the ISO/IEC 15909 [17]. Here, instead, we use the simpler Colored Petri Nets (CPNs), inspired by [18]. Finally, our compositions will only have at most two participants, instead of any number of participants as in the general version. This means for instance that we only allow to compose two transitions in the same composition, but not three. As our three behavioral operators are associative, it is trivial to extend these binary compositions to n-ary, but doing so increases the complexity of the notation. The interested reader will fin a complete formal definition in [24].

5.1. General notations

Over the years, colored Petri nets have beed defined numerous times, with slight variations. We consider that the basic structures (typing, multisets, assignments, etc.) do not need to be fully redefined once again. Thus, we will only give some basic notations for these mathematical objects, without giving a full definition.

In the following, $C = \{c_1, c_2, ..., c_n\}$ is a finite set of type names. With a slight abuse of language, the set of elements of a type c_i is also denoted by the name c_i itself, and we note $C = \cup_{i\in[1..n]}c_i$. We assume that the set of booleans $\mathbb{B}$ is always included in C. Let us for instance consider the set $C = \{\mathbb{B}, \mathbb{N}\}$, a set with two types: the booleans and the naturals. Here, $\mathbb{N}$ denotes the name of the type but also the full set of values of that type, $\mathbb{N} = \{0, 1, 2, \ldots\}$.

We say that a set S is indexed by C if each element of S is associated with a unique element of C, and we note S_c where $c \in C$ the set of elements of S indexed by c. For instance, we consider the set of variables $X = \{\$\mathrm{b}, \$\mathrm{x}, \$\mathrm{y}\}$ where $X_{\mathbb{B}} = \{\$\mathrm{b}\}$ and $X_{\mathbb{N}} = \{\$\mathrm{x}, \$\mathrm{y}\}$. Here, each variable is typed, i.e., it is associated with a type name. In the following, every set of variables X will always be indexed by some set of type names C. Note that for simplicity reasons we do not consider multi-typed systems.

Each type defines a set of values, but it also defines *expressions*. These expressions are built with constants (the values), operators, and variables. The set expressions of a type built with a set of variables X is noted $expr(c_i, X)$. The set of *closed* expressions (expressions without variables) of a type c_i is noted $expr(c_i, \emptyset)$. For instance, we have $\{5, 1 + 1, \$\mathrm{x}{+}3\} \subset expr(\mathbb{N}, X)$, and $\{5, 1 + 1\} \subset expr(\mathbb{N}, \emptyset)$. Trivially, we have $c_i \subseteq expr(c_i, \emptyset) \subseteq expr(c_i, X)$. Each type defines an evaluation function $eval$ such that each expression $e \in expr(c_i, \emptyset)$ is evaluated to some element of c_i, noted $eval(expr) \in c_i$. Again, we will not go into the details of this evaluation (which varies from formalism to formalism), and we will simply assume that this evaluation exists.

An assignment σ is a function $X \rightarrow C$ where $\forall c_i \in C,\ \forall x \in X_{c_i},\ \sigma(x) \in c_i$ (each variable is assigned a value of the adequate type). We trivially extend an assignment to the full set of expressions. Assigning a value to each variable of an expression results in a closed expression: if σ is an assignment of X, we have $\forall e \in expr(c_i, X), \sigma(e) \in expr(c_i, \emptyset)$.

A fundamental notion in the domain of Petri nets is the *multiset* (also called *bag*). A multiset is a set-like structure where each element can appear multiple times. We note $[c_i]$ the set of multisets of elements of c_i, and $[C] = \cup_{i\in[1..n]}[c_i]$ (note that this means that all the elements in a multiset must have the same type). Similarly, the set of multisets of *expressions* of a type c_i with the set of variables X is noted $[expr(c_i, X)]$. We assume that the application of the assignments is also defined for multisets, and we get $\forall mult \in [expr(c_i, X)],\ \sigma(mult) \in [expr(c_i, \emptyset)]$, where $mult$ is a multiset of expressions in $[c_i]$ and

$\sigma(mult)$ is the application of an assignment to the multiset, resulting in a multiset of closed expressions. Moreover, we also assume that the evaluation functions are extended to the multisets from $[expr(c_i, \emptyset)]$. We get for any substitution σ the following result: $\forall mult \in [expr(c_i, X)],\ eval(\sigma(mult)) \in [c_i]$.

5.2. Syntactic definitions

To formalize the semantics of the LCM, we need of course to give a basic syntactic formal definition. As mentioned earlier, this definition is an extremely simplified version of the full LLAMAS formalism. The full definition is to be found in [24].

We will use here a slightly modified version of the CPNs as defined by Jensen in [18]. Our variant adds two features to the Petri nets as defined by Jensen, parametric transitions and a distinction between active and passive transitions.

Definition 5.1. (Petri net)
A Petri net is a tuple $N = \langle C, X, P, T, M_0 \rangle$ where:

- C is a finite set of types, called *color sets*
- X is a finite set of variables indexed by C
- P is a finite set of places indexed by C
- T is a set of transition axioms as defined below. We define $T = T_{act} \uplus T_{pass}$ where T_{act} is called the set of *active* transitions and T_{pass} the *passive* transitions.
- M_0 is a marking of the net as defined below, called the initial marking of N.

The markings of a Petri net mentioned in def. 5.1 are defined in def. 5.2, and the transitions in def. 5.3.

Definition 5.2. (Marking of a Petri net)
Let $N = \langle C, X, P, T, M_0 \rangle$ be a Petri net. The set of markings of N, noted $Mark(N)$, is the set of functions $P \rightarrow [C]$ where $\forall c_i \in C, \forall p \in P_{c_i}, M_0(P) \in [c_i]$ (each place can only hold tokens of the corresponding color). We also define $M_0 \in Mark(N)$.

Definition 5.3. (Transition)
A transition is a tuple $\langle name, Par, G, In, Out \rangle$ where:

- $name$ is the name of transition
- $Par \in X^*$ is an ordered list of formal parameters
- G is boolean expression from $expr(C, X)$ (i.e., $eval(\sigma(G)) \in \mathbb{B}$ for any substitution σ), representing the guard of the transition
- In is the set of pre-conditions of the transition, defined as a function $P \rightarrow [expr(C, X)]$ where $\forall c_i \in C,\ \forall p \in P_{c_i},\ In(p) \in [expr(c_i, X)]$
- Out is the set of post-conditions of the transition, defined as a function $P \rightarrow [expr(C, X)]$ where $\forall c_i \in C,\ \forall p \in P_{c_i},\ Out(p) \in [expr(c_i, X)]$

Remember that it is possible to apply a substitution and an evaluation to any multiset from the set $[expr(c_i, X)]$ and get a multiset from $[c_i]$. Thus, it is easy to transform the pre and post conditions from def. 5.3 into markings as defined in def. 5.2 by using a substitution σ. Formally, the following inclusion holds for any transition $\langle name, Par, G, In, Out\rangle$ in a Petri net N and any substitution σ: $eval(\sigma(In)), eval(\sigma(Out)) \in Mark(N)$.

Example 5.4. (Petri net)
Consider the example from Fig. 5. We consider that this figure contains two separate Petri nets, joined by a composition c11 (which we will define later). Both Petri nets are defined over the natural numbers, i.e., we have $C = \{\mathbb{B}, \mathbb{N}\}$. We assume that both Petri nets manipulate the same color sets and the same set of variables $X = X_{\mathbb{N}} = \{\$x, \$y, \$a\}$.

The first Petri net (that we will call net1, even though this name does not appear in the figure) has one place p1 indexed by $\mathbb{N}$, and its initial marking is $M_0(p1) = [2]$, i.e., the place p1 contains one token of value 2. net1 also has a transition t1. This transition has one formal parameter, $x, and its guard is empty (i.e., it is equal to true). Its precondition is defined as $In(p1) = [\$x]$, and its post-condition as $Out(p1) = []$. The transition t1 is passive, and thus we have $T_{act} = \emptyset$ and $T_{pass} = \{t1\}$ in net1. The second Petri net (net2) has a similar definition. It has one place p2 indexed by $\mathbb{N}$ whose initial marking is empty ($M_0(p2) = []$), and one transition t2, with one formal parameter $y and an empty guard. The precondition of this transition is defined as $In(p2) = []$, and its post-condition as $Out(p2) = [\$y]$. Again, the transition t2 is passive.

At this point, out definitions are similar to the classical definitions in the domain of Petri nets, with two simple additions, the parameters in the transitions and the distinction between active and passive transitions. We will now get into the details of the definitions specific to our modular formalism. We will start by defining how to compose a set of Petri nets by means of *compositions*.

From now on, we will consider a set of Petri nets as defined in def. 5.1, noted $Nets = \{N_i\}_{i\in[1..n]}$. We assume that all these Petri nets are defined over the same color sets C and the same variables X. Note that, unlike [20], we do not consider modularity of the data types used by the different Petri nets. We also assume that the sets of places and transitions from the different Petri nets are disjunct, no place nor transition can belong to two different Petri nets. The formal definition of the compositions is given in def. 5.5.

Definition 5.5. (Composition)
Let $Nets = \{N_i\}_{i\in[1..n]}$ be a set of *disjoint* Petri nets. We define a *composition* over $Nets$ as a tuple $\langle name, Par, G, Op, Parts\rangle$ where:

- $name$ is the name of the composition
- $Par \in X^*$ is an ordered list of formal parameters
- G is boolean expression from $expr(C, X)$ (i.e., $eval(\sigma(G)) \in \mathbb{B}$ for any substitution σ), representing the guard of the composition
- $Op \in \{$merge, any, sequence, not, read$\}$ is a composition operator
- *Parts* is one (if $Op \in \{$not,read$\}$) or two (if $Op \in \{$merge,any,sequence$\}$) *participant expression(s)*. A participant expression is noted $ev(p_1, ..., p_n)$ where:

- ev is an event, i.e., ev is either an expression $N_i.t$ where t is the name of a transition of the Petri net N_i, or the name of some composition over $Nets$ (note the recursivity of this definition)
- $(p_1, ..., p_n) \in expr(C, X)^*$ is a list of effective parameters of the event compatible with its formal parameters. This means that if $(x_1, ..., x_m)$ are the formal parameters of the event, then $n = m$ and each variable x_i has the same type as the corresponding expression p_i.

In def. 5.5 we defined that the events in a composite Petri net are the transitions in the individual Petri nets and the compositions between these Petri nets. Both the transitions as defined in def. 5.3 and the compositions as defined in def. 5.5 have a list of formal parameters. Thus, the last condition in def. 5.5, i.e., the correspondence between the formal and the effective parameters of the events, is equally applicable to the transitions and the compositions.

Example 5.6. (Composition)
Consider again the example from Fig. 5. There are two Petri nets in this figure, called net1 and net2, and one composition whose name is c11. c11 has one formal parameter $a, an empty guard, and it uses the operator merge. This composition has also two participant expressions, the first one is noted net1.t1($a), and the second one net2.t2($a+1). Note that the effective parameter of the first (resp. second) participant, the expression $a (resp. $a+1), is compatible with the formal parameter of the transition t1 (resp. t2), which was the variable $x (resp. $y). Thus, following Def. 5.5, the composition c11 is defined by the tuple $\langle$c11, ($a), true, merge, (net1.t1($a), net2.t2($a+1))$\rangle$.

We have now defined the syntax of our Petri nets and their compositions. The next step is to define composite systems, called *composite Petri nets*, defined in def. 5.7. Please note that this version of composite Petri nets is very simple: there is no notion of encapsulation nor hierarchy. Only the most basic features needed to define the behavior of the LCM are defined.

Definition 5.7. (Composite Petri net)
A composite Petri net is a tuple $CN = \langle C, X, Nets, Comps \rangle$ where:

- C and X are respectively color sets and a set of variables, similarly to what was defined in def. 5.1
- $Nets = \{N_i\}_{i \in [1..n]}$ is a set of disjoint Petri nets defined over C and X
- $Comps$ is a set of compositions over $Nets$ as defined in def. 5.5. Similarly to the transitions in def. 5.1, we define $Comps = Comps_{act} \uplus Comps_{pass}$. $Comps_{act}$ is the set of *active* compositions, and $Comps_{pass}$ the set of *passive* compositions.

Back in def. 5.2 we defined the markings of a single Petri net. The markings of a composite Petri net CN is trivially built as the disjunct union of the individual markings of each Petri net in CN. This is more precisely defined in def. 5.8.

Definition 5.8. (Markings of a composite Petri net)
Let $CN = \langle C, X, Nets, Comps \rangle$ be a composite Petri net, where $Nets = \{N_i\}_{i \in [1..n]}$ and each net $N_i = \langle C, X, P_i, T_i, M_{0_i} \rangle$. Let $P_{CN} = \uplus_{i \in [1..n]} P_i$ be the set of all places in the individual Petri nets (remember that the sets of places are disjoint). A marking of CN is a function $M_{CN} : P_{CN} \to [C]$ where $\forall c \in C, \forall p \in (P_i)_c, M_{CN}(p) \in [c]$.

5.3. Semantics

In the previous section we defined syntactically the notion of composite Petri net, a collection of Petri nets connected by means of compositions. In this section we will define the behavior of these composite Petri nets by building their LTSs, i.e., a set of triplets $\langle \text{marking}, \text{event}, \text{marking} \rangle$ representing their evolution.

Traditionally, the firing of a transition t from a marking m in a Petri net is defined by the removal of some resources (its pre-condition noted $\text{Pre}(t, m)$) and the addition of other resources (its post-condition noted $\text{Post}(t, m)$). This is usually noted $m' = m - Pre(t, m) + Post(t, m)$. To define the behavior of our compositions, we will manipulate the pre and post-conditions of the events that are composed. Thus, we will build our LTS in two steps: we will first define the Pre- and Post-conditions of the events in a structure called $PrePost$, and then we will use these Pre- and Post-conditions to build the LTS.

Definition 5.9. (*$PrePost$*)
Consider a composite net CN as defined in def. 5.7. $PrePost_{CN}$ is a set of tuples $\langle m, ev, Pre, Post \rangle$, where m is a marking of CN, ev is an event in the Petri net (see def. 5.5), Pre and $Post$ are markings meant to represent the Pre and Post conditions of the event from marking m. In this structure, Pre and $Post$ are *closed* markings, i.e., they represent the actual resources that will be removed and added from the current marking, after substituting every variable.

Each element in $PrePost_{CN}$ represents a possible firing of an event ev from a marking m. As in any HLPN formalism, there may be multiple ways to fire the same event from some marking. This means that, for the same marking m and the same event ev, there may be multiple tuples $\langle m, ev, Pre, Post \rangle$ in $PrePost_{CN}$. In fact, in almost every case, the $PrePost_{CN}$ structure will be infinite as there may be an infinite number of ways to fire some event. Note that $PrePost_{CN}$ contains every possible firing in a Petri net, including the firings from markings that are not reachable from the initial state. In the following, as there is no ambiguity, we will simply note $PrePost$ rather than $PrePost_{CN}$.

To define this $PrePost$ structure, we will use some inference rules shown in Figs. 8 to 16. We consider a composite Petri net $CN = \langle C, X, Nets, Comps \rangle$, where $Nets = \{N_i\}_{i \in [1..n]}$ and each $N_i = \langle C, X, P_i, T_i, M_{0_i} \rangle$. The first of these inference rules, shown in Fig. 8, handles the case of the transitions of the Petri nets in N_i. The events defined by these transitions are noted $N_i.\,\text{t}(\text{EffPar})$, where t is the name of a transition in N_i and EffPar is a list of *closed* effective parameters, i.e., a list of expressions from $expr(C, \emptyset)$ compatible with the formal parameters of the transition.

$$\frac{\begin{array}{c}\langle \text{t}, \text{Par}, \text{G}, \text{In}, \text{Out} \rangle \in T_i \\ \sigma \in Subst(C, X) \\ \sigma(Par) = eval(\text{EffPar}) \\ eval(\sigma(G)) = true \\ \text{Pre} = eval(\sigma(\text{In})) \\ \text{Post} = eval(\sigma(\text{Out})) \\ \text{Pre} < m\end{array}}{\langle m, N_i.\,\text{t}(\text{EffPar}), \text{Pre}, \text{Post} \rangle \in PrePost}$$

Figure 8: $PrePost$ for transitions

The first premise of this rule checks that there is a transition in the net N_i with the adequate name. In the second premise we consider some substitution σ, and in the third premise we check that this substitution follows the structure of the effective parameters of the transition. For instance, if the transition considered had two formal parameters, ($\$x$,$\$y$), and there was two effective parameters, ($1+1$, 3), then the substitution σ must ensure that $\sigma(\$x) = 2$ and $\sigma(\$y) = 3$. Note that for $eval$ to be applicable on EffPar, the latter must be composed solely of *closed* expressions. The fourth premise checks that the guard of the transition is respected by the substitution σ. Note that there may be multiple (or even an infinite number) of substitutions that satisfy these conditions. Given a substitution σ, the next two premises compute the concrete pre- and post-conditions of the transition, by applying the substitution and then evaluating the resulting expressions. Finally, the last premise checks that the transition is enabled, i.e., that the computed pre-conditions are included in the marking m. This pattern will be repeated in the next inference rules: any element from $PrePost$ denotes an event that is *enabled* (i.e., it can be fired) from the marking m. The result of the inference rule is a tuple from $PrePost$, where we have computed concrete pre and post conditions for a given transition with given effective parameters.

Attentive readers may notice that we use a small abuse of notation, as technically Pre and $Post$ in this inference rule are markings of the individual Petri net N_i, while m is a marking of the composite Petri net. As the set of places in the Petri nets N_i are disjoint, we can trivially extend a marking of an individual Petri net to the whole composite Petri net by associating empty multisets to the places that belong to the other Petri nets.

Example 5.10. (Transition firings)
Let us go back to the example from Fig. 5. The marking m in this figure is defined by $m(\mathrm{p1}) = [2]$ and $m(\mathrm{p2}) = []$. Let us try to fire the transition $\mathrm{t1}(1+1)$, where $1+1$ is the effective parameter of the transition. Notice that we can use complex expressions as effective parameters for firing the events, and these complex expressions must be evaluated during the computation of the firing. For this, we try to build a tuple $\langle m, \mathrm{net1}\,.\,\mathrm{t1}(1+1), \mathrm{Pre}, \mathrm{Post}\rangle$ in $PrePost$. Let us follow the inference rule from Fig. 8. The first step is to determine the precise structure of $\mathrm{t1}$. We built this structure in ex. 5.4, where we determined that the transition $\mathrm{t1}$ was defined by the tuple $\langle \mathrm{t1}, (\$x), \mathrm{true}, [\$x], []\rangle$. The next step in the inference rule is to consider some substitution σ. Any substitution could be considered here. The next step checks that $\sigma((\$x)) = eval((1+1))$, and thus $\sigma(\$x) = 2$. Any substitution that assigns any other value to $\$x$ would be discarded by this condition. The next step is to check if the guard is evaluated to true with the given substitution. As the guard is empty, it is trivially true for any substitution. The next two steps compute the pre- and post-conditions of the firing, by defining $\mathrm{Pre}(\mathrm{p1}) = eval(\sigma([\$x])) = eval[2] = [2]$ and $\mathrm{Post}(\mathrm{p1}) = eval(\sigma([])) = eval([]) = []$. The resulting tuple from $PrePost$ is $\langle m, \mathrm{net1}\,.\,\mathrm{t1}(1+1), [2], []\rangle$.

The next five inference rules will compute the elements of $PrePost$ for the compositions. Let us start with the merge operator in Fig. 9. The first five premises are similar to the rule from ex. 8, i.e., we check that a composition with the given name exists, we consider a substitution that matches the formal and the effective parameters and we check that the substitution validates the guard. The next two premises check that the two events that are composed with the merge operator are individually enabled, and they extract their pre and post conditions (resp. Pre_1, Post_1, Pre_2 and Post_2). Note that the eventual variables in both events are substituted with σ, and thus all the expressions passed as parameters of these events are closed. In the next two premises we build the pre-(resp. post-)conditions of the composition itself by adding the preconditions (resp. postconditions) of both events. Finally, the last premise from

$$\frac{\begin{array}{c}\langle \mathrm{c}, \mathrm{Par}, \mathrm{G}, \text{merge}, (\mathrm{ev}_1, \mathrm{ev}_2)\rangle \in Comps \\ \sigma \in Subst(C, X) \\ \sigma(Par) = eval(\mathrm{EffPar}) \\ eval(\sigma(G)) = true \\ \langle m, \sigma(ev_1), \mathrm{Pre}_1, \mathrm{Post}_1\rangle \in PrePost \\ \langle m, \sigma(ev_2), \mathrm{Pre}_2, \mathrm{Post}_2\rangle \in PrePost \\ \mathrm{Pre} = \mathrm{Pre}_1 + \mathrm{Pre}_2 \\ \mathrm{Post} = \mathrm{Post}_1 + \mathrm{Post}_2 \\ \mathrm{Pre} < m\end{array}}{\langle m, \mathrm{c}(\mathrm{EffPar}), \mathrm{Pre}, \mathrm{Post}\rangle \in PrePost}$$

Figure 9: $PrePost$ for merge

this rule checks that the preconditions of this composition are met by the marking, which ensures that the resulting composition is enabled.

The next rule we consider is represented in Fig. 10. It computes the pre and post conditions of the compositions that use the operator any. This rule is quite simple, we check if we can compute the pre and post conditions of any of the two events ev_1 or ev_2, and return the result found. Note that contrary to the other $PrePost$ inference rules, we do not need to check if the global pre-condition of this composition is included in the considered marking. Indeed, if the internal event ev_i was enabled, it follows that the whole composition has enough ressources. We only need to check that the guard of the composition is evaluated to *true*.

$$\frac{\begin{array}{c}\langle \mathrm{c}, \mathrm{Par}, \mathrm{G}, \text{any}, (\mathrm{ev}_1, \mathrm{ev}_2)\rangle \in Comps \\ \sigma \in Subst(C, X) \\ \sigma(Par) = eval(\mathrm{EffPar}) \\ eval(\sigma(G)) = true \\ i \in \{1, 2\} \\ \langle m, \sigma(\mathrm{ev}_i), \mathrm{Pre}, \mathrm{Post}\rangle \in PrePost\end{array}}{\langle m, \mathrm{c}(\mathrm{EffPar}), \mathrm{Pre}, \mathrm{Post}\rangle \in PrePost}$$

Figure 10: $PrePost$ for any

Our next rule correspond to the operator sequence and it is represented in Fig. 11. This rule is illustrated by an example in ex. 5.11 which considers the net from Fig. 12.

Example 5.11. (Firing a sequence)
The inference rule from Fig. 11 is illustrated by the example in Fig. 12. This net has five places, two passive transitions and an active composition c13. For simplicity reasons, we only consider black tokens, without any parameter nor guard on the events. The marking in Fig. 12 is noted $\{1, 0, 0, 1, 0\}$ for places

$\{\text{p1,p2,p3,p4,p5}\}$. Let us apply the rule from Fig. 11 to compute the firing of c13. The first step is to define c13 by the tuple $\langle \text{c13}, (), \text{true}, \text{sequence}, (\text{net.t1}, \text{net.t2})\rangle$ where net is the name of our Petri net. Let us skip the substitution and guard, which as we mentioned are irrelevant here, and jump to the fifth premise of the inference rule. This premise considers a $PrePost$ tuple for the first event, net.t1, fired from marking m. By applying the rule in Fig. 8, we get $\langle m, \text{net.t1}, \{1,0,0,0,0\}, \{0,1,1,0,0\}\rangle$. The sixth premise asks to compute $m' = m - Pre_1 + Post_1$, which is the marking that would be reached if we fired net.t1 from m. The result is $m' = \{0,1,1,1,0\}$. The seventh premise computes the $PrePost$ tuple for firing ev_2, i.e., net.t2, from m'. With the rule from Fig. 8, we get the tuple $\langle m', \text{net.t2}, \{0,0,1,1,0\}, \{0,0,0,0,1\}\rangle$. Note that the rule in Fig. 8 not only computes the pre and post-conditions of both transitions, it also checks that they are enabled from their respective markings.

$$\frac{\begin{array}{c}\langle \text{c}, \text{Par}, \text{G}, \text{sequence}, (\text{ev}_1, \text{ev}_2)\rangle \in Comps \\ \sigma \in Subst(C, X) \\ \sigma(Par) = eval(\text{EffPar}) \\ eval(\sigma(G)) = true \\ \langle m, \sigma(ev_1), \text{Pre}_1, \text{Post}_1\rangle \in PrePost \\ m' = m - Pre_1 + Post_1 \\ \langle m', \sigma(ev_2), \text{Pre}_2, \text{Post}_2\rangle \in PrePost \\ \text{Pre} = \text{Pre}_1 + (\text{Pre}_2 - \text{Post}_1) \\ \text{Post} = \text{Post}_2 + (\text{Post}_1 - \text{Pre}_2) \\ \text{Pre} < m\end{array}}{\langle m, \text{c}(\text{EffPar}), \text{Pre}, \text{Post}\rangle \in PrePost}$$

Figure 11: $PrePost$ for sequence

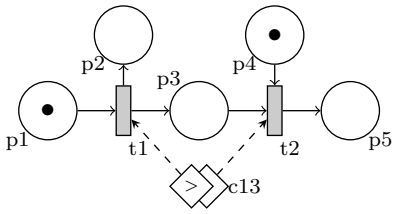

Figure 12: Sequence example.

We have now computed $Pre_1 = \{1,0,0,0,0\}$, $Post_1 = \{0,1,1,0,0\}$, $Pre_2 = \{0,0,1,1,0\}$, and $Post_2 = \{0,0,0,0,1\}$. The next two premises in our inference rule give us the global pre and post-conditions of the composition, $Pre = \{1,0,0,1,0\}$ and $Post = \{0,1,0,0,1\}$. Thus, the composition c13 removes a token from p1 and p4, and it adds a token to p2 and p5. During the computation of c13, p3 received a token and immediately lost it, and thus in the final result it appears as if p3 had not been involved in the firing. The last premise in the rule from Fig. 11 checks if the composition is enabled from m by checking that the pre-condition is included in m, which is indeed the case in this example. The resulting tuple from this example is $\langle \{1,0,0,1,0\}, \text{c13}, \{1,0,0,1,0\}, \{0,1,0,0,1\}\rangle$.

Finally, Fig. 13 and Fig. 14 show the inference rules to compute the pre and post conditions of the compositions that use respectively the observers read and not. The rule for read in Fig. 13 simply checks if the event in the composition is enabled from the marking m. The pre and postconditions of the resulting composition are empty. This means that even if the composition is enabled, its firing will have no effect on the current marking. The rule in Fig. 14 for the not operator is visually almost identical, but it has heavy theoretical implications. This rule checks that there does *not* exist a substitution σ such that the event ev_1 is executable. This negation is indicated by the operator $\sim$. Computing this kind of negation may be undecidable in some cases, as we will mention in Section 5.4.

$$\frac{\begin{array}{c}\langle \mathrm{c}, \mathrm{Par}, \mathrm{G}, \mathrm{read}, (\mathrm{ev}_1)\rangle \in Comps \\ \sigma \in Subst(C, X) \\ \sigma(Par) = eval(\mathrm{EffPar}) \\ eval(\sigma(G)) = true \\ \langle m, \sigma(\mathrm{ev}_1), \mathrm{Pre}, \mathrm{Post}\rangle \in PrePost\end{array}}{\langle m, \mathrm{c}(\mathrm{EffPar}), [], []\rangle \in PrePost}$$

Figure 13: $PrePost$ for read

$$\sim \left(\frac{\begin{array}{c}\langle \mathrm{c}, \mathrm{Par}, \mathrm{G}, \mathrm{not}, (\mathrm{ev}_1)\rangle \in Comps \\ \sigma \in Subst(C, X) \\ \sigma(Par) = eval(\mathrm{EffPar}) \\ eval(\sigma(G)) = true \\ \langle m, \sigma(\mathrm{ev}_1), \mathrm{Pre}, \mathrm{Post}\rangle \in PrePost\end{array}}{\langle m, \mathrm{c}(\mathrm{EffPar}), [], []\rangle \in PrePost}\right)$$

Figure 14: $PrePost$ for not

At this point we have explained in detail the inference rules that allow to build the $PrePost$ structure, which contains the set of pre and post-conditions of all the events in composite Petri net. From this structure, it is very simple to build LTS of the composite net. This is done in Fig. 15 and Fig. 16.

$$\frac{\begin{array}{c}\langle \mathrm{t}, \mathrm{Par}, \mathrm{G}, \mathrm{In}, \mathrm{Out}\rangle \in T_{i\ act} \\ \mathrm{EffPar} \in expr(C, \emptyset)^* \\ \langle m, N_i.\,\mathrm{t}(\mathrm{EffPar}), \mathrm{Pre}, \mathrm{Post}\rangle \in PrePost \\ m' = m - \mathrm{Pre} + \mathrm{Post}\end{array}}{\langle m, N_i.\,\mathrm{t}(\mathrm{EffPar}), m'\rangle \in LTS}$$

Figure 15: LTS for transitions

$$\frac{\begin{array}{c}\langle \mathrm{c}, \mathrm{Par}, \mathrm{G}, Op, Part\rangle \in Comps_{act} \\ \mathrm{EffPar} \in expr(C, \emptyset)^* \\ \langle m, \mathrm{c}(\mathrm{EffPar}), \mathrm{Pre}, \mathrm{Post}\rangle \in PrePost \\ m' = m - \mathrm{Pre} + \mathrm{Post}\end{array}}{\langle m, \mathrm{c}(\mathrm{EffPar}), m'\rangle \in LTS}$$

Figure 16: LTS for compositions

Example 5.12. (LTS)
Back in ex. 5.11 we built an element of $PrePost$ for the example in Fig. 12 for the composition c13. Let us now consider the inference rule from Fig. 16. The first premise considers some *active* composition in the composite Petri net with name c. In our case, the name of the composition is c13. The second premise considers a list of *closed* effective parameters for the composition. In our example, the list of effective parameters is empty as the composition does not define any formal parameter. The third premise considers a tuple from $PrePost$ with the composition c from marking m. As mentioned above, we consider the tuple $\langle\{1,0,0,1,0\}$,c13, $\{1,0,0,1,0\}, \{0,1,0,0,1\}\rangle$ from ex. 5.11. Note that our inference rules for the $PrePost$ structure already ensured that the composition was enabled from this marking with these Pre and Post conditions. The fourth premise in the inference rule in Fig. 16 computes the successor marking m', by stating $m' = m - \mathrm{Pre} + \mathrm{Post} = \{1,0,0,1,0\} - \{1,0,0,1,0\} + \{0,1,0,0,1\} = \{0,1,0,0,1\}$. Based on these premises, the inference rule concludes that $\langle\{1,0,0,1,0\}$,c13, $\{0,1,0,0,1\}\rangle$ belongs to the LTS of our composite Petri net.

We have now a mathematical definition of the LTS of a composite Petri net. This definition may include infinite domains, a negation on these infinite domains (for the not operator) and infinite computations (in the case of infinite recursions). Given these characteristics, it is legitimate to wonder if our LTS is actually well defined from a theoretical point of view. Moreover, even if the LTS is mathematically sound, we could still wonder if it is possible to compute it. The next section will attempt to answer these questions. First, we will state the conditions on which our LTS and $PrePost$ structures actually exist, and then we will mention some restrictions that allow us to actually compute these structures.

5.4. Computational aspects

In the previous sections we defined the semantics of the LCM by means of inference rules. It is important to discuss the computational issues that we may encounter when applying these rules. Indeed, in some cases, trying to compute a composition may lead to an infinite calculation. The first cause of this problem is the presence of infinite domains, particularly for the operator not. To compute this operator, we must ensure that there does *not* exist *any* substitution that allows the composed event to be executed. If we consider infinite domains, we may need to check an infinite number of substitutions, which is undecidable in the general case. The second cause for infinite computations is recursivity. Indeed, as in any recursive framework, a careless user may define compositions whose computation may be infinite, or even be inconsistent. For instance, one could define a composition $\mathrm{c} = \mathrm{not}(\mathrm{c})$ whose computation is infinite.

From a theoretical point of view, the problem of infinite computations could be avoided by applying the Closed-World Assumption (CWA) [28]. Simply put, this mechanism allows to infer a negative from a lack of information: if we could not prove that an event is or is not enabled, then we assume that it is not enabled by default. In this case, the composition c mentioned earlier would fail as its computation would be infinite. Of course, from an implementation point of view, CWA is undecidable. A weaker version of CWA is Clark's Negation As Failure (NAF) [8], as implemented in Prolog, the language used in our prototype. NAF also infers negative statements but only for events that can be proved to fail in finite time. For infinite computations, NAF does not return a result, and their execution most likely leads to a program error. This is the case for our composition c mentioned above.

To avoid this problem, we could consider two different techniques. The first one is stratification. Simply put, asking for a set of compositions to be stratified amounts to impose some restrictions on their definitions in order to ensure finite computations. In the most basic versions of stratification [10], we would avoid any recursive composition and severely restrict the use of the not operator. More recent advances in this domain such as [31] are more flexible. While stratification is an effective method to handle recursivity and negation, its power is limited in the case of infinite domains for the variables in the compositions. To avoid the problem of infinite domains, our prototype implementation of LLAMAS uses algebraic unfolding [14] as in our model checking tool AlPiNA [15, 5]. In general terms, we ask the user to provide a bound for any infinite domain, and we only compute values up to that bound. If the model encounters a value beyond the bound set by the user, an error message is returned to the user. Please note that only algebraic unfolding is implemented in our prototype, while stratification remains an interesting work perspective.

The problem of transactional events with infinite computations may seem a big leap from traditional Petri nets, where all the events are finitely computed. Nevertheless, there are in fact various variants of Petri nets that share this characteristic. Among them, we can cite the reference nets [22], CO-OPN/2 [2], the Hornets [19] or the zero-safe Petri nets [3]. As our objective is to cover as many Petri nets formalisms as possible, it is not a surprise that we share these "infinite computations" with the most advanced formalisms from the literature.

6. The LLAMAS Language - An Example

In the previous sections we explained in detail the composition mechanism of LLAMAS. The objective of this section is to introduce the rest of the formalism. For simplicity and space issues, we will only present the language by means of an example. A complete formalization is given in [24].

6.1. Internal Behavior of a Module - Petri Net

We begin the description of our example with the Petri net from Fig. 17. Together with the compositions we will give in the next section, this Petri net will represent a bounded counter. It has two places named Bound and Counter. The initial value of the token in place Bound is a variable, and thus its actual value will be determined at instantiation (see Section 6.4). There are also three transitions named GetBound, Reset, and Inc. All these transitions are passive (see Section 4.2). Notice that, if it was an active transition, an execution of GetBound would remove the token from place Bound. We will see in the next section that this behavior will be restrained by the compositions. All three transitions have one parameter that unifies with the value taken from the respective place.

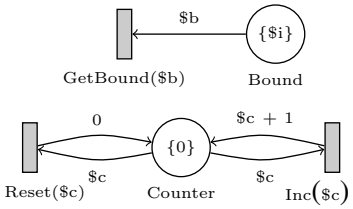

Figure 17: Petri net example.

6.2. Compositions

Fig. 18 shows four compositions that have been added to the example in Fig. 17. The composition in the center is called ReadBound. It uses the observer read, whose symbol is $\odot$, and it binds the transition GetBound. The read operator means that when the composition ReadBound is executed, it checks if transition GetBound can be executed and it unifies the variable $b with the value taken from place Bound, without any modification to the marking of the net (see Section 4.3). Composition ReadBound binds GetBound (see Section 4.1), which means that GetBound cannot participate in any other composition. Thus, even though an execution of ReadBound removes the token from place Bound, we ensured by means of the composition ReadBound that this token will never actually be removed. The second composition, called Reset, *binds* the transition Reset, and *calls* the composition ReadBound, and uses the merge operator. When it is executed, both composition ReadBound and transition Reset will be executed simultaneously, as long as the guard of the composition ($c = $b) is evaluated to *true*. Thus, the composition Reset checks if the counter has reached its bound and resets it to 0. Composition Inc works in a similar way. It increases the value of the counter if it has not reached the value of the bound. The final composition, Tick, binds both compositions Reset and Inc with the operator any. Whenever Tick is executed, one composition between Reset and Inc is also executed, but not both. If neither Reset nor Inc are enabled, neither is Tick. All these compositions are passive and thus, for now, our example does not include a single executable event.

6.3. Interface - Basic LLAMAS Module

The notion of encapsulation is central in most modular formalisms. It implies the existence of some elements in a module that are available to other modules (the interface), and some elements that are internal to the module. Some modular Petri nets formalisms define some places and/or transitions as the interface of modules. Other formalisms define entirely new elements to define the interface. The composition mechanism we defined in Section 4 requires that we adopt the latter case. Indeed, following

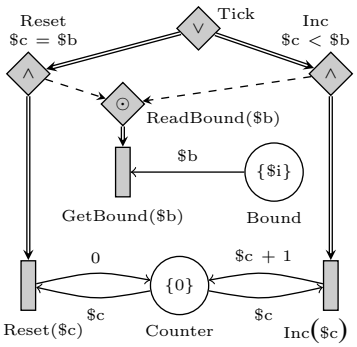

Figure 18: Compositions example.

the definition of the compositions in the LCM, we need interface elements that allow us to carry *calls* and *bindings* across modules. Moreover, as the compositions are directed, we need to distinguish between input elements and output elements (we respectively call them *services* and *requests*). Thus, we have four kinds of elements in our module interfaces: binding and non-binding services and binding and non-binding requests. Fig. 19 illustrates the four possible combinations. It shows two transitions and one composition (resp. t1, t2 and c2) inside a module and two transitions and one composition (resp. t3, t4 and c1) outside the module. The interface of the module contains a binding and a non-binding service (resp. s1 and s2), represented by triangles directed towards the interior of the module. It also contains a binding and a non-binding request (resp. r2 and r1), represented by triangles directed towards the environment. c1 binds t1 by means of s1, and thus t1 is not available to other compositions, including compositions inside the module. c1 also calls t2 (by means of s2) and t3. Similarly, c2 binds t4 and calls t3 and t2. The distinction between input and output interface elements can be found in many formalisms, see for instance the import/export interfaces from [20], or the distinction between sockets and ports in [16]. Services and requests in a LLAMAS interface are parametric, similarly to the transitions and compositions.

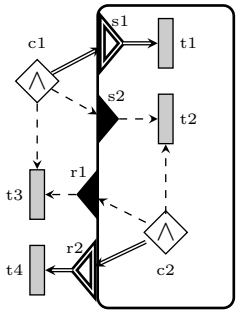

Figure 19: A simple example of interface.

Fig. 20 shows an interface added to the example from Fig. 18. Together, a Petri net, a set of compositions, and an interface form a non-hierarchical LLAMAS module. The module from Fig. 20 is called $BCounter$, and it takes one parameter $i to set the initial value of place Bound. The interface of $BCounter$ defines a non-binding service called Tick and a non-binding request called Overflow. Service Tick calls composition Tick, thus making this composition available to the container. Composition Reset

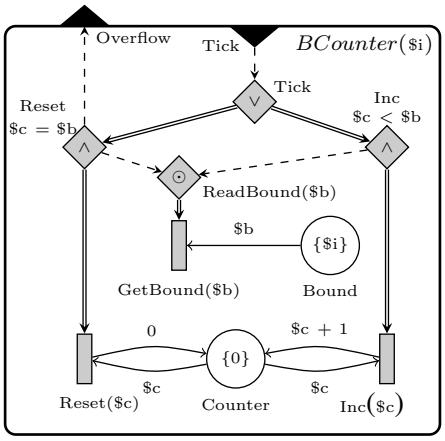

Figure 20: A LLAMAS module.

now calls the request Overflow, apart from calling ReadBound and binding transition Reset. Thus, whenever there is an overflow, not only will the counter be reseted, but a notification will be sent to the container of $BCounter$. If the container is not able to respond to this request, the composition Reset will fail. The next section shows an example of such container.

6.4. Hierarchy and Static Instantiation

Another important feature of modular formalisms is hierarchy. In hierarchical Petri nets a module (called the *container*) may encapsulate other modules (called *submodules*), which in turn may encapsulate their own submodules. The hierarchy is a partial order relation between the modules in a system. Each container defines how its submodules communicate, both between themselves and with the container. A hierarchical LLAMAS module is thus composed of a Petri net, an interface, a set of submodules and a set of compositions (notice the similarity with our general definition in Section. 3.3). The hierarchical mechanism we use in LLAMAS was inspired by Modular PNML [20]. In Modular PNML, each module defines a set of submodule sockets, and each submodule socket is denoted only by an interface, which defines the synchronization contract between the container and the submodule.

Before showing an example, let us mention one last important feature of LLAMAS: instantiation. An instantiation mechanism facilitates the definition of various identical (or similar) modules. It means that the modules in a system are copies of an initial definition, a blueprint. These copies may be fine tuned when they are created. Following the vocabulary of object-oriented languages, we call this blueprint a *class*, and its copies *instances* of the class. The instantiation may be *static*, i.e., performed once during the initial definition of the system, or *dynamic*, i.e., instances may be created during its execution.

Fig. 21 shows a hierarchical module that uses three static instances of the module $BCounter$ from Fig. 20 to model a clock. The module $Clock$ contains three submodules sockets, all of them with the same interface as $BCounter$. The first submodule on the left (the hours) has a bound of 23, and the two others (minutes and second respectively) have a bound of 59. The module $Clock$ defines a composition called Second, which is active. It is the first and only active event we encounter in the whole example (remember that all the transitions and compositions in $BCounter$ were passive). Each time Second is executed, it calls the service Tick from the rightmost submodule (the seconds). Whenever this submodoule has an overflow

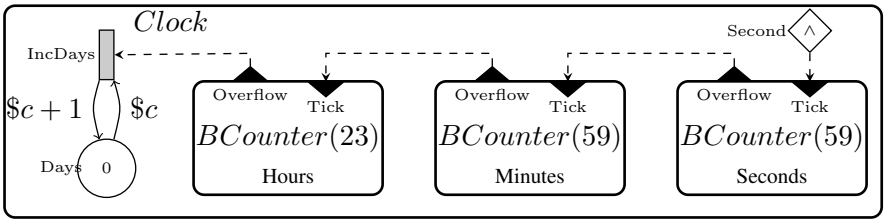

Figure 21: A hierarchical LLAMAS module representing a clock.

(i.e., 60 seconds have passed), it calls the service Tick from the module in the center, by means of its Overflow request. Similary, whenever there is an overflow on the center module (60 minutes), the service Tick from the leftmost submodule is called. Finally, whenever this submodule calls its own request Overflow (24 hours), the transition IncDays is called, which increases the counter in place Days. The module $Clock$ has clearly an infinite number of reachable states. If we remove the place Days but keep the transition IncDays, the module will have $24 \times 60 \times 60 = 86400$ reachable states, without any deadlock (executing Second from the last state would reset the whole system). If we also remove the transition IncDays the request Overflow from the leftmost submodule will not be executable, and this submodule would be blocked when its counter would reach the value 23. The whole system would then have the same 86400 states, but the last one, with marking $\langle 23, 59, 59 \rangle$, would be a deadlock.

6.5. Dynamic Instantiation

In this section we have seen a basic introduction to LLAMAS by means of an example. Some features of the language could not be described here for space reasons. The most important one is a reference-based implementation of the NWN paradigm [32], which adds dynamic instantiation to LLAMAS. This and other secondary features are included in the formal definition of the language.

7. Case Study: Modular PNML

LLAMAS was created with the intent of being able to handle most if not all the modular mechanisms from the Petri nets literature. We will show in this section one example of formalism whose composition mechanism can be translated to LLAMAS, Modular PNML [20]. We chose this example because Modular PNML, as mentioned previously, is an important candidate to serve as semantic platform in the approach we presented in Section 2. Moreover, Modular PNML handles modularity of the data types, which is out of the scope of LLAMAS. Thus, it is interesting to see the relation between the two formalisms.

Let us first briefly describe our example of Modular PNML. Details and a formal definition of the language are to be found in [20]. The composition mechanism of Modular PNML is a particular example of fusions of places and transitions. The module interfaces are composed of places and transitions of the Petri nets, which are either *imported* or *exported* by the modules. Modular PNML is a hierarchical formalism, and thus the modules can embed other modules. For instance consider the module on the left side of Fig. 22. The top layer of this module represents its interface. There is one exported transition called t1, and one exported place called p2. This Modular PNML module contains a submodule. The interface of this submodule is represented in the bottom section of the module. This interface shows

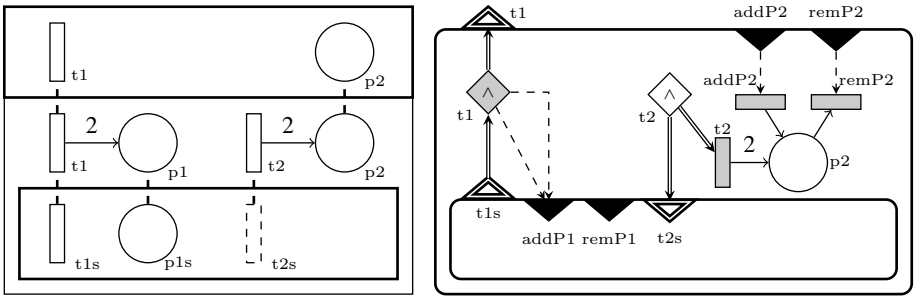

Figure 22: A Modular PNML example and its LLAMAS translation.

that the submodule exports one transition called $t1s$ and one place called $p1s$, and it imports the transition $t2s$. To distinguish between exported and imported elements, the latter are graphically represented by dashed figures. In our example, the container "uses" the elements $t1s$ and $p1s$ that are exported by the submodule, and the submodule "uses" the transition $t2$ from the container. The elements $t1s$, $p1s$ and $t2s$ in the submodule are respectively merged to $t1$, $p1$ and $t2$ in the container. These merges are graphically represented by dashed lines. Note that the container re-exports the transition $t1$ which was originally exported by the submodule. Please note also that, even though our example does not show it, Modular PNML defines a static instantiation mechanism. This is not seen in the example because we only show the interface of the submodule but not its implementation.

We show the equivalent LLAMAS module in the right hand side of Fig. 22. There is a different representation for the sharing of transitions and the sharing of places. The LCM of LLAMAS is an event-based mechanism, which makes the representation of the sharing of transitions rather straightforward. Consider the case of the transition $t1s$ which is exported by the submodule. Whenever the submodule executes this transition, it must inform the container of this execution. This is done by means of a binding *request* called $t1s$. Similarly, the imported transition $t2s$ is represented by *service* $t2s$, and the export of $t1$ in the interface of the container is represented by request $t1$.

Sharing places is a different problem. In Section 4, we claimed that even though the LCM is an event-based formalism, it is able to simulate state-based mechanisms. In our example, the submodule exports the place $p1s$. This means that the submodule is the owner of $p1s$, and it is offering its container the possibility to add and remove tokens from this place. We represent this with two services called $addP1$ and $remP1$. Thus, the submodule is the only one who has a real access to his place $p1s$. When the container wants to interact with this place, it must request access to the submodule. Similarly, the container also exports one place called $p2$. This means that it is giving access to this place to its own container. This is represented again by two services, $addP2$ and $remP2$, that call transitions to add and remove tokens from $p2$. Notice that our example is a simple P/T Petri net. The translation of HLPN places can be achieved in the LCM by means of parametric services and requests. Note that Modular PNML does not allow to share values with interface transitions, and the authors of [20] mentioned this as a possible improvement. This is already defined in LLAMAS, again by using parametric services and requests.

Up until now we described the translation of the module interfaces, let us now mention the translation of the internal Petri nets. First, only the local places are represented in the LLAMAS module, the foreign places will be accessed by means of the corresponding services and requests. In our example, $p1$ belongs to the submodule, and thus it does not appear in the container. On the other hand, $p2$ is a local place, and it remains in the result of the translation.

Each transition from Modular PNML is represented by two different artifacts in LLAMAS. First, all the interactions of the transitions with other modules (imports, exports, and access to foreign places) are represented as a composition. Second, the local behavior of the transition is implemented as a transition in LLAMAS. In our example, transition $_{t1}$ in Modular PNML adds two tokens to $_{p2}$, which is a foreign place. In fact, $_{t1}$ does not have any local behavior in the container. This is why there is no transition called $_{t1}$ in the resulting LLAMAS module, only a composition. Transition $_{t1}$ adds two tokens to the foreign place $_{p1}$, this is represented in LLAMAS by two simultaneous calls to the service $_{addP1}$.
On the other hand, $_{t2}$ has a local behavior, as it adds two tokens to the local place $_{p2}$. In the LLAMAS module, this is represented by the transition called $_{t2}$. Moreover, $_{t2}$ is shared with the submodule. This is represented in LLAMAS by a binding from the composition $_{t2}$ to the binding service $_{t2s}$. Note that all the compositions in Modular PNML are fusions of places and transitions, which is translated by the use of the operator $_{merge}$ in every LLAMAS composition.

Because of readability and space reasons, we cannot include other use cases in this paper. Nevertheless, we considered complete translations of many formalisms, including the Hierarchical CP-nets [16], CO-OPN/2 [2], the reference nets [22], the Object Petri nets [23], the Petri Box and M-nets families [9, 21] and others. Stepping out of LLAMAS central objective, we also considered composition mechanisms from formalisms such as the ITS [29] and CSP [13].

8. Verification of LLAMAS models

The objective of LLAMAS is to provide a common ground to define the semantics of modular Petri nets formalisms. This would improve the understanding and communication of formalisms and computational techniques in the scientific community. For this to be effective, LLAMAS must not only be an expressive enough semantic platform, it must be a complete modeling formalism on its own. To show that this is the case, we have developed a small prototype implementation of LLAMAS, available as an Eclipse plugin at `http://goo.gl/BgONM`. This prototype allows to create LLAMAS models and, for small models, to compute their state space. We plan to extend this tool to implement efficient verification of modular models by using the recent advances in model checking with hierarchical Decision Diagrams (DDs). DDs are data type structures specially designed to encode big sets of states. Recent years have seen the development of hierarchical variants of DDs such as the Σ Decision Diagrams (ΣDDs) [4], which were inspired by [30]. Our model checking tool AlPiNA [15, 5] uses these structures to check properties on nets with sometimes over 10^{230} states.

In ΣDDs, the states of a Petri net are encoded as data structures designed to save memory. The behavior of the events is encoded by means of operations on these structures called *homomorphisms*. To encode a Petri net transition as a ΣDD homomorphism, we define a set of small operations that are combined with three operators that are built-in in the ΣDD framework: the union, the composition and the intersection (details are to be found in [4]). Let us consider an example. Let $t1$ be a transition that takes one token from a place $p1$ and adds two tokens to a place $p2$. A simplified version of the homomorphism that encodes the behavior of $t1$ can be noted $H_{t1} = H^+(p2, 2) \circ H^-(p1, 1)$, where the H^+ operation represents a post-condition, the H^- a pre-condition and the $\circ$ is the ΣDD composition operation. Consider a transition $t2$ encoded as $H_{t2} = H^+(p3, 1) \circ H^-(p4, 2)$. The LLAMAS merge of these transitions would be encoded as $H_{\mathrm{merge}(t1,t2)} = H^+(p2, 2) \circ H^+(p3, 1) \circ H^-(p1, 1) \circ H^-(p4, 2)$.

In short, in the ΣDD framework, the structure of the models and their behavior are defined by sepa-

rate artifacts. Because of this, this framework is well adapted to represent event-based formalisms such as LLAMAS. Moreover, the execution of complex events is encoded as the combination of small operations by means of operators. This matches the mindset of the LCM, where complex events are defined as smaller events combined with five operators. Finally, the ΣDD support recursive and parametric executions. Because of these similarities, an implementation of LLAMAS in the ΣDD framework is a promising endeavor.

9. Conclusion and perspectives

In this article we proposed an approach to create a unified definition process of modular extensions of Petri nets. With this approach new formalisms can have their syntax and semantics defined in the context of a common framework. For the definition of the semantics we propose a new formalism, called LLAMAS, specifically designed to include the characteristics of most if not all the existing modular Petri nets formalisms. To this day, we have not encountered a composition mechanism in the Petri nets literature that could not be translated to the LCM, even if sometimes the translation is not trivial. We believe that our approach would improve the understanding and communication of novel modeling and verification techniques in the scientific community.

This article only partially showed the work we developed in LLAMAS. Among the elements that had to be left out, let us cite the metamodel to standardize syntactic elements that we mentioned in Section 2, a complete formalization of the language that includes the NWN mechanism for dynamic instantiation, and other case studies than the one presented in Section 7. The main novelty in this article with relation to our previous works is a simplified formal definition of the composition mechanism in LLAMAS, given in Section 5.

We think that LLAMAS has promising perspectives. In Section 8 we briefly sketched some ideas about the verification of LLAMAS models using Decision Diagrams, an important research field. Efficient model checking of LLAMAS models is clearly the most important research perspective in this field. On the other hand, the language itself can be extended to include even more modular considerations, such as inheritance and subtyping relations between modules or, as one of our anonymous reviewers suggested, allowing to define active and passive events at instantiation time. Finally, while the current implementation of LLAMAS is a textual prototype, we plan to implement a graphical version of the language and integrate it with our tool AlPiNA.

References

[1] Al-Shabibi, A., Buchs, D., Buffo, M., Chachkov, S., Chen, A., Hurzeler, D.: Prototyping Object Oriented Specifications, in: *Applications and Theory of Petri Nets 2003* (W. Aalst, E. Best, Eds.), vol. 2679 of *LNCS*, Springer Berlin Heidelberg, 2003, ISBN 978-3-540-40334-0, 473–482.

[2] Biberstein, O., Buchs, D., Guelfi, N.: Object-Oriented Nets with Algebraic Specifications: The CO-OPN/2 Formalism, in: *Concurrent Object-Oriented Programming and Petri Nets* (G. Agha, F. Cindio, G. Rozenberg, Eds.), vol. 2001 of *LNCS*, Springer Berlin Heidelberg, 2001, ISBN 978-3-540-41942-6, 73–130.

[3] Bruni, R., Montanari, U.: Zero-Safe Nets, or Transition Synchronization Made Simple., *Electronic Notes in Theoretical Computer Science: Proceedings of EXPRESS'97, 4th workshop on Expressiveness in Concurrency*, 7, Elsevier Science, 1997.

[4] Buchs, D., Hostettler, S.: Sigma Decision Diagrams, *TERMGRAPH 2009: Preliminary proceedings of the 5th International Workshop on Computing with Terms and Graphs* (A. Corradini, Ed.), number TR-09-05 in TERMGRAPH workshops, Università di Pisa, 2009.

[5] Buchs, D., Hostettler, S., Marechal, A., Risoldi, M.: AlPiNA: An Algebraic Petri Net Analyzer, in: *Tools and Algorithms for the Construction and Analysis of Systems* (J. Esparza, R. Majumdar, Eds.), vol. 6015 of *LNCS*, Springer Berlin Heidelberg, 2010, ISBN 978-3-642-12001-5, 349–352.

[6] Christensen, S., Damgaard Hansen, N.: Coloured Petri Nets extended with channels for synchronous communication, in: *Application and Theory of Petri Nets 1994* (R. Valette, Ed.), vol. 815 of *LNCS*, Springer Berlin Heidelberg, 1994, ISBN 978-3-540-58152-9, 159–178.

[7] Christensen, S., Petrucci, L.: Modular Analysis of Petri Nets, *Comput. J.*, **43**(3), 2000, 224–242.

[8] Clark, K. L.: Negation as failure, in: *Logic and Data Bases* (J. Minker, Ed.), vol. 1, Plenum Press, 1978, 293–322.

[9] Devillers, R., Klaudel, H., Riemann, R.-C.: General parameterised refinement and recursion for the M-net calculus, *Theoretical Computer Sc.*, **300**, 2003, 259–300, ISSN 0304-3975.

[10] Gelder, A. V.: Negation as failure using tight derivations for general logic programs, *The Journal of Logic Programming*, **6**(1–2), 1989, 109 – 133, Special Issue:Third {IEEE} Symposium on Logic Programming.

[11] Hillah, L., Kordon, F., Petrucci, L., Trèves, N.: PNML Framework: An Extendable Reference Implementation of the Petri Net Markup Language, in: *Applications and Theory of Petri Nets* (J. Lilius, W. Penczek, Eds.), vol. 6128 of *LNCS*, Springer Berlin Heidelberg, 2010, ISBN 978-3-642-13674-0, 318–327.

[12] Hillah, L.-M., Kordon, F., Lakos, C., Petrucci, L.: Extending pnml Scope: A Framework to Combine Petri Nets Types, in: *Transactions on Petri Nets and Other Models of Concurrency VI* (K. Jensen, W. Aalst, M. Ajmone Marsan, G. Franceschinis, J. Kleijn, L. Kristensen, Eds.), vol. 7400 of *LNCS*, Springer Berlin Heidelberg, 2012, ISBN 978-3-642-35178-5, 46–70.

[13] Hoare, C. A. R.: Communicating sequential processes, *CACM*, **21**(8), 1978, 666–677.

[14] Hostettler, S.: *High-Level Petri Net Model Checking, The Symbolic Way*, Ph.D. dissertation, University of Geneva, 2012.

[15] Hostettler, S., Marechal, A., Linard, A., Risoldi, M., Buchs, D.: High-Level Petri Net Model Checking with AlPiNA, *Fundam. Inf.*, **113**(3-4), August 2011, 229–264, ISSN 0169-2968.

[16] Huber, P., Jensen, K., Shapiro, R.: Hierarchies in coloured petri nets, in: *Advances in Petri Nets 1990* (G. Rozenberg, Ed.), vol. 483 of *LNCS*, Springer Berlin Heidelberg, 1991, ISBN 978-3-540-53863-9, 313–341.

[17] ISO/IEC: *Software and Systems Engineering – High-level Petri Nets, Part 1: Concepts, Definitions and Graphical Notation*, International Standard ISO/IEC 15909-1, 2004.

[18] Jensen, K.: *Coloured Petri nets: basic concepts, analysis methods, and practical use*, Number vol. 1 in EATCS monographs on theoretical computer science, Springer-Verlag, 1992.

[19] Köhler-Bußmeier, M.: Hornets: Nets within Nets Combined with Net Algebra, in: *Applications and Theory of Petri Nets* (G. Franceschinis, K. Wolf, Eds.), vol. 5606 of *LNCS*, Springer Berlin Heidelberg, 2009, ISBN 978-3-642-02423-8, 243–262.

[20] Kindler, E., Petrucci, L.: Towards a Standard for Modular Petri Nets: A Formalisation, in: *Applications and Theory of Petri Nets* (G. Franceschinis, K. Wolf, Eds.), vol. 5606 of *LNCS*, Springer Berlin Heidelberg, 2009, ISBN 978-3-642-02423-8, 43–62.

[21] Klaudel, H., Pommereau, F.: M-nets: a survey, *Acta Informatica*, **45**(7-8), 2009, 537–564.

[22] Kummer, O.: *Referenznetze*, Logos Verlag Berlin, 2002, ISBN 9783832500351.

[23] Lakos, C.: Object Oriented Modeling with Object Petri Nets, *Concurrent Object-Oriented Programming and Petri Nets*, 2001.

[24] Marechal, A.: *Unifying the Syntax and Semantics of Modular Extensions of Petri Nets*, Ph.D. dissertation, University of Geneva, 2013.

[25] Marechal, A., Buchs, D.: *Modular extensions of Petri Nets: a generic template metamodel*, Technical Report 220, University of Geneva, 2012.

[26] Marechal, A., Buchs, D.: *Modular extensions of Petri nets: a survey*, Technical Report 218, University of Geneva, 2012.

[27] Marechal, A., Buchs, D.: Unifying the Semantics of Modular Extensions of Petri Nets, in: *Application and Theory of Petri Nets and Concurrency* (J.-M. Colom, J. Desel, Eds.), vol. 7927 of *LNCS*, Springer Berlin Heidelberg, 2013, ISBN 978-3-642-38696-1, 349–368.

[28] Reiter, R.: On Closed World Data Bases, *Logic and Data Bases*, 1977.

[29] Thierry-Mieg, Y., Poitrenaud, D., Hamez, A., Kordon, F.: Hierarchical Set Decision Diagrams and Regular Models, *15th Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS, Springer, 2009, ISBN 978-3-642-00767-5.

[30] Thierry-Mieg, Y., Poitrenaud, D., Hamez, A., Kordon, F.: Hierarchical Set Decision Diagrams and Regular Models, in: *Tools and Algorithms for the Construction and Analysis of Systems* (S. Kowalewski, A. Philippou, Eds.), vol. 5505 of *LNCS*, Springer Berlin Heidelberg, 2009, ISBN 978-3-642-00767-5, 1–15.

[31] Tiu, A.: Stratification in Logics of Definitions, in: *Automated Reasoning* (B. Gramlich, D. Miller, U. Sattler, Eds.), vol. 7364 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2012, ISBN 978-3-642-31364-6, 544–558.

[32] Valk, R.: Object Petri Nets, in: *Lectures on Concurrency and Petri Nets* (J. Desel, W. Reisig, G. Rozenberg, Eds.), vol. 3098 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2004, ISBN 978-3-540-22261-3, 819–848.