Verifying Reference Nets By Means of Hypernets: a Plugin for Renew

Marco Mascheroni[1], Thomas Wagner[2], and Lars Wüstenberg[2]

[1] Dipartimento di Informatica, Sistemistica e Comunicazione
Università degli Studi di Milano Bicocca
Viale Sarca, 336, I-20126 Milano (Italy)**
mascheroni@disco.unimib.it
[2] University of Hamburg,
Faculty of Mathematics, Informatics and Natural Sciences,
Department of Informatics
http://www.informatik.uni-hamburg.de/TGI/

Abstract. In this paper we examine ways to verify reference nets, a class of high level Petri nets supported by the Renew tool. We choose to restrict reference nets to hypernets, another nets-within-nets model more suitable for verification purposes thanks to an expansion toward 1-safe Petri nets. The contribution of the paper is the implementation of such analysis techniques by means of a Renew plugin. With this plugin it is now possible to draw, and to analyze a hypernet. The work is demonstrated by means of a simple example.

Keywords: Verification, High-level Petri nets, Reference nets, Hypernets

1 Introduction

The verification of properties of a software system has become an important part of software engineering. Especially specifications critical to the correct execution of a software system need to be verified in order to guarantee them after deployment. The problem of verification is its complexity and the effort required for it. Without proper methods the verification itself is difficult, costly and time-consuming.

In this paper we approach the general problem of verification with the help of Petri nets. The formalism is deeply rooted within established theoretical and formal methodologies, as well as being supported by a multitude of tools and analysers. Petri nets have been studied in detail and contain properties, for which established verification techniques exist. Using Petri nets the general approach is to map and translate specific software issues and properties to these Petri

** Partially supported by MIUR, and DAAD net properties. These properties can then be verified using the known Petri net techniques. Assertions made for these can then be translated back for the software behind it.

High level nets, Petri net models enriched with additional abstraction constructs, are well suited to represent complex systems due to their high abstraction constructs. One of their problems is that verification of their properties is difficult. Properties which are computable with low-level formalisms become undecidable, and thus cannot be verified anymore in some high-level models. However, high-level formalisms can be restricted in some way so that they can be translated into low-level formalisms, which in turn can be verified again. In particular, the interest of this paper is on high level nets which use the nets-within-nets paradigm, formalisms in which the tokens of a Petri nets can be structured themselves as Petri net. The two formalisms analyzed are *reference nets*, the formalism used as a basis for the Renew tool, and *hypernets*, another nets-within-nets formalism with particular restrictions that allow the expansion toward an equivalent 1-safe Petri nets. In this paper we will show how to translate a subset of the high-level reference net formalism into hypernets, which in turn can be easily translated into 1-safe nets. These can then be analysed by existing toolsets. The main result of our work is the implementation of a Renew hypernet plugin which incorporates features for computing S-invariants, and features for model checking a hypernet. As far as we know, this is the first time that analysis techniques typical of Petri nets has been implemented in a tool which support the nets-within-nets paradigm, and it is mature enough to be used in a real application context. In the rest of the paper when we will talk about invariants we are always referring to S-invariants.

The paper is structured into the following sections. Following this introduction the theoretical and technical background is shortly discussed in section 2. This section will focus on the reference net and hypernet formalisms. In section 3 we will show how to translate reference nets into hypernets and determine the prerequisites for analysis. Section 4 describes the Hypernet plugin created for Renew. Section 5 gives a short example how these different tools are incorporated and used to analyse a given net. The conclusion of the paper is found in section 6.

2 Background and related work

In this section we will introduce by means of examples the basic theoretical formalisms used in this paper, as well as motivate why we have chosen them as our means of verification and modelling. The interested reader can find them in the cited references. In general Petri nets offer a simple way of modelling concurrent behaviour of a system. Higher level nets often introduce abstractions from the simple net models, which offer structures and methods not available to or difficult to model in low-level Petri nets. One major such abstraction is the idea of nets within nets, introduced in [11] for Object Petri nets. This paradigm allows for arbitrary nets to be the tokens of other nets. In this way it is possible to model the behaviour and interaction of different entities within a complex system, all modelled with Petri nets. Using these formalisms to model and even implement software systems is quite natural. Of course high-level Petri nets and especially formalisms following the nets-within-nets idea are far more complex then the relatively simple low-level Petri nets. This increases the effort and complexity of verifying properties within these nets, which is the main motivation of this paper.

In the following subsections we will describe the reference and hypernet formalisms.

2.1 Reference Nets and Renew

The reference net formalism serves as the starting point of our examinations. It was described in [7]. It is a high level Petri net formalism based on the nets-within-nets paradigm. In this formalism it is possible for tokens within a net to be almost arbitrary objects and especially other Petri nets. Nets can then be used like tokens within their respective so-called system net, but it is also possible to let nets of different layers communicate with one another. The reference net formalism uses reference semantics. This means that tokens within a net do not exclusively correspond to their object/net (value semantics), but only reference their object/net. As a result of this multiple tokens can refer to the same object. This makes it possible to express complex systems in a natural way.

Communication between different net instances within the reference net formalism is handled via synchronous channels, based on the concepts proposed in [5]. Synchronous channels connect two transitions during firing. Transitions inscribed with a synchronous channel can only fire synchronously, meaning that both transitions involved have to be activated before firing can happen. During firing arbitrary objects can be transmitted bidirectionally over the channel. While the exchange of data is bidirectional there is a difference in the handling of the two transitions. The transition, or more accurately the inscription of the transition, initiating the firing is called the *downlink*. The downlink must know the name of the net in which the other transition, the so-called *uplink*, is located. The inscription of the downlink has the form *netname:channelname(parameters)*, in which the parameters are the objects being send and received during firing. If the downlink calls an uplink located in the same net the net name is simply replaced by the keyword *this*. The uplink's inscription is similar, but looses the net name, so that it has the form *:channelname(parameters)*. Uplinks are not exclusive to one downlink and can be called from multiple downlinks, so that this construct can be used in a flexible way. It is also possible to synchronise transitions over different abstraction levels. While during firing one downlink is always linked to just one uplink, it is possible to inscribe one transition with multiple downlinks, so that more than two transitions can fire simultaneously.

Figure 1 shows a simple example of a reference net system. The example was modelled using the Renew tool, which will be described later. It models a producer/consumer system, which holds an arbitrary number of producers and consumers. The system consists of three kinds of nets: the system net, the

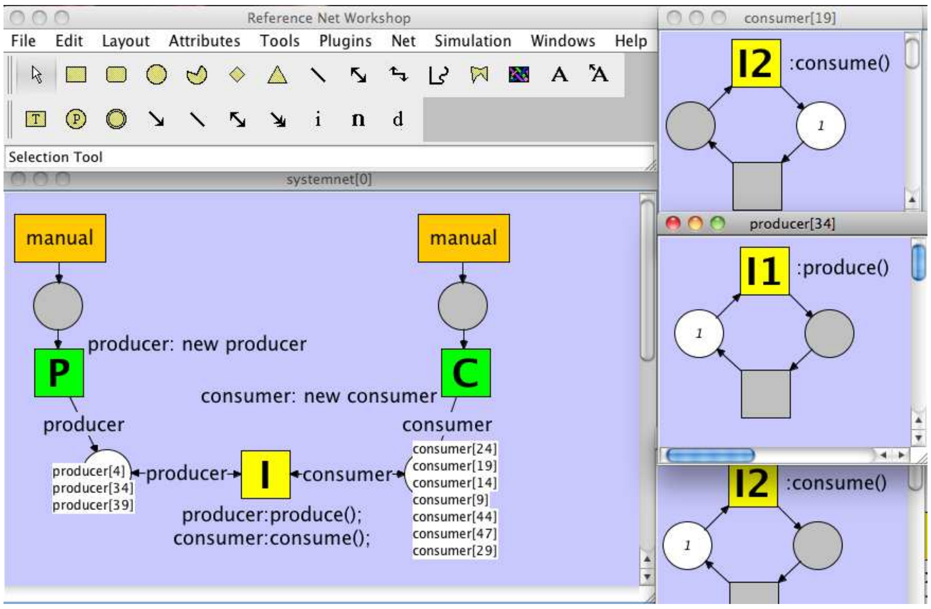

Fig. 1. Reference net example

producer nets and the consumer nets. The producer and consumer nets both possess the same basic structure, but use different channels. The system net serves as a kind of container for the other nets. The transitions labeled **manual** initiate the creation of new producers and consumers by creating new tokens when a user manually fires them during simulation[3]. The transitions labeled C and P actually create new producer or consumer nets when firing. These new nets are put onto the places below the transitions. The transition labeled I synchronises the firing of a transition in one consumer and one producer each (labeled I1 and I2 in the other nets). In this way it is possible to simulate the behaviour in such a way, that whenever a producer produces a product an arbitrary consumer consumes it. It is of course possible to enhance this model by, for example, adding an intermediary storage, which can store items from arbitrary producers until consumers need them. Another way of making the model more realistic is to explicitly model the products as nets as well. That way they would not just be simple tokens but actual objects being exchanged via the synchronous channels between the producers and consumers. In this case the parameters of the channels would be the nets, which would be transmitted from within the producer nets into the consumer nets.

The Petri net editor and simulator Renew (The **RE**ference **NE**t **W**orkshop) was developed alongside the reference net formalism, and is also described in [7] as well as in [8]. It features all the necessary tools needed to create, modify, simu-

[3] This is a special function of the Renew tool, which was used for this example.

late and examine Petri nets of different formalisms. It is predominantly used for reference nets, but can be enhanced and extended to support other formalisms. It is fully plugin based, meaning that all functionality is provided by a number plugins that can be chosen, depending on the specific needs. Plugins can encapsulate tools, like a file navigator or certain predefined net components, or extensions to the standard reference net formalism, like hypernets or workflow nets. Renew is freely available online and is being further developed and maintained by the Theoretical Foundations Group of the Department for Informatics of the University of Hamburg. Since the tool supports the idea of nets within nets and is flexible enough to support multiple formalisms, it was chosen as the basic environment for the examinations of this paper.

2.2 Hypernets

As we will discuss later in section 3, we introduce hypernets in this paper because they have been used as a restriction of the reference nets formalism to allow property verification in Renew. Hypernets are a nets-within-nets formalism introduced to model systems of mobile agents [2]. After their introduction several studies has been conducted on hypernets. In [3] it has been shown that it is possible to expand a hypernet in a 1-safe Petri net in such a way that the (hyper) reachability graph of the hypernet is equivalent to the reachability graph of the 1-safe net. In [1] a class of transition system, called agent aware transition systems, has been introduced to describe the behaviour of hypernets. In order to model a class of membrane systems, a generalisation of the hypernet formalism which relaxes some constraints of the basic formalism was introduced under the name of generalised hypernet in [4], and a theorem proving the existence of an expansion towards 1-safe nets for generalised hypernets was proved in [9].

Due to technical limitations in the Renew tool only the basic version of the formalismi [3] has been implemented. Now we will informally discuss how hypernets work by means of an example. From a structural point of view a hypernet is a collection of (possibly empty) agents $\mathcal{N} = \{A_1, A_2, ..., A_n\}$, which are modelled as particular Petri nets. A state of a hypernet is obtained associating to each one of the A_i agents (nets), but one, a place p belonging to one of the other agents. That place will be the place which contains the agent A_i. This containment relation induces a hierarchical structure which by definition must be a tree. The root of the tree is the only agent which is not associated to any place (this agent is the system net).

The system evolves moving agents from place to place. A peculiar characteristic of hypernets is that the hierarchical structure is not static, but an agent can be moved from a place p belonging to an agent A_i, to a place q belonging to a distinct agent A_j. Another characteristic of hypernets is that agents cannot be created or destroyed. To ensure this ”law of conservation of tokens” each net representing an agent is structured as a set of *modules* which have the structure of synchronised state machines, enriched with some *communication* places that allow the exchange of tokens between two agents close in the hierarchy. Agents and modules have a sort, and an agent can only travel along modules of the same sort.

In Figure 2, and Figure 3 the hypernet modelling a slightly modified version of the *one seater plane* case of study is drawn. This case of study has been introduced in [3], and models an airport in which planes can do basic things like landing, deplaning/boarding passenger, refuelling, and taking off. The changes we made in regards to the number of travellers in the example, the simplification of the safety refuel check and the part of the hypernet which makes sure a plain is empty when it is being refuelled.

To keep the example simple we considered a version with a plane which has only one seat. We choose to illustrate this example to show in an informal way how hypernets works. Moreover, in Section 5 we will show how it is possible to prove some properties of this simple example using the Renew plugin we developed.

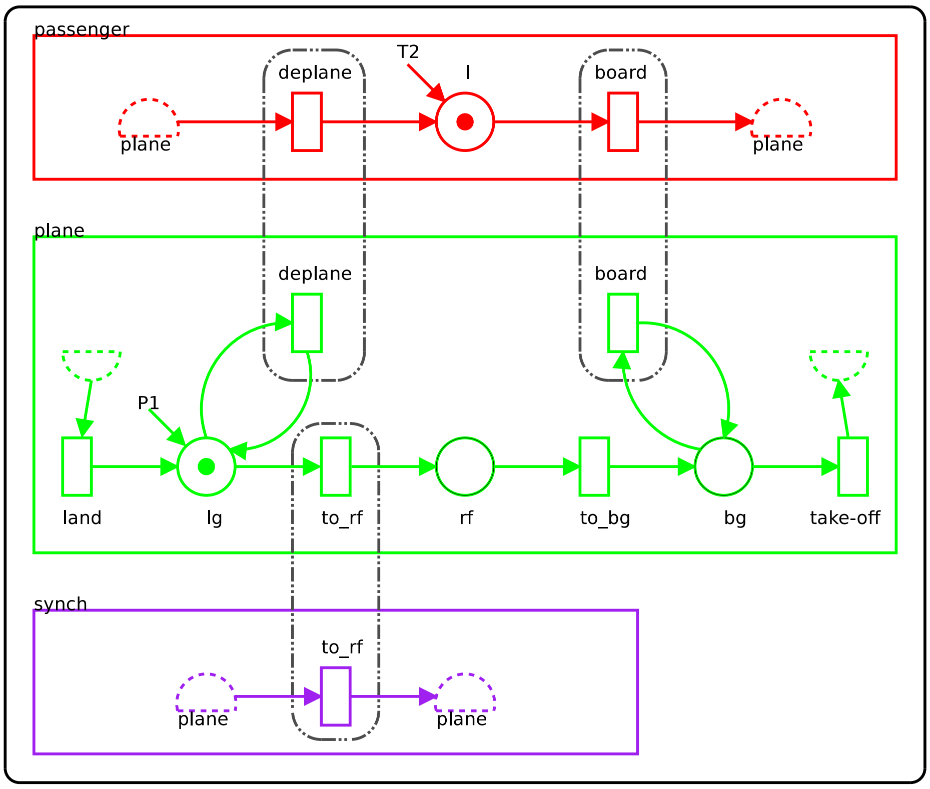

Fig. 2. Airport agent

The agent in Figure 2 models the behaviour of the airport. It has three modules, one for handling passengers, one for handling planes and one for synchronisation purposes. Transition *board* belongs to both module *passenger* and module *plane*, and can only be executed synchronously. The same applies for transitions *deplane* and *to_rf*. Communication places are the dashed half circles. They can either be *up-communication places*, used for communicating with the net at the level immediately above in the hierarchy (such as the two communicating places of the module plane in the airport agent), or *down-communication places*, used to communicate with an agent located in another module of the current net (such as the communication places in the synch, and passenger modules of the airport). In the latter case, a name of a module is provided. In this module there must be an agent ready to provide the *traveling* token which will be moved in the hierarchy, otherwise the transition is not enabled.

For example, transition *deplane* of the passenger module in Figure 2 has an input communication place which indicates that a token is expected. Since this communication place is marked with the *plane* annotation, the traveling token which is being moved to place l must be provided by a plane agent. This plane agent must be located in the input place of transition deplane in module plane of the airport, namely lg. In the example the only agent which can provide a token is $P1$. The traveling token, which must be a passenger, is then selected

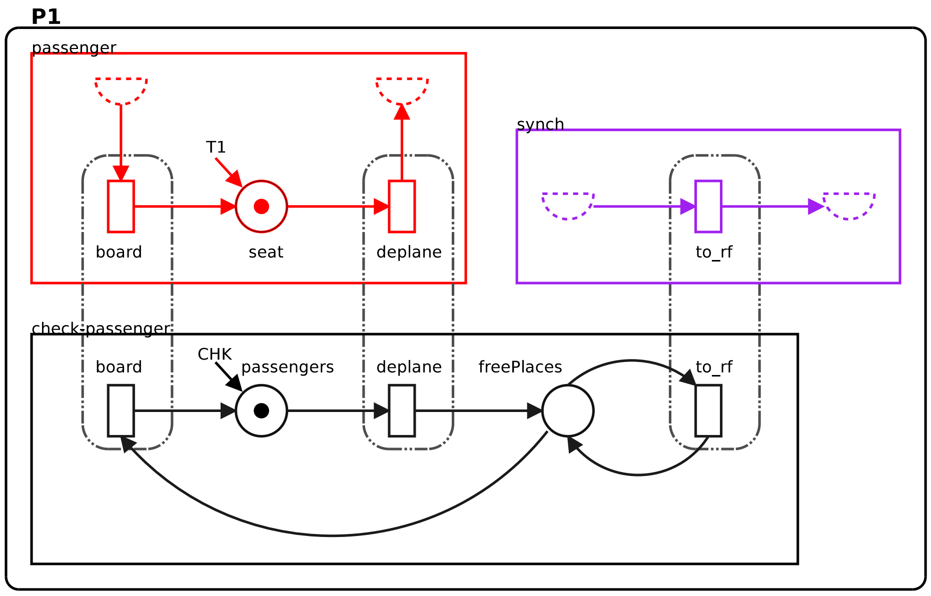

Fig. 3. The $P1$ plane agent shown in Figure 2

and taken from the *seat* place of the plane agent (Figure 3), and moved to l.

Transition *to_rf* is another example of use of communication places. From the airport perspective it is only required that an agent located in the *plane* module has a module *synch* containing with a transition *to_rf* preceeded and followed by two up-communication places. This requirement is fulfilled by agent $P1$, but from the $P1$ perspective it is also required the enabledness of the synchronized *to_ rf* transition in the module check-passenger. Therefore this configuration *to_ rf* is not enabled because *freePlaces* is not marked.

Hypernet being a high level net model means that the execution of a transition, like *deplane*, has several *firing-modes* [10]. Each firing-mode in a hypernet is a called *consortium*, and is obtained by selecting a transition, a set of agents that contain the transition, and a set of *passive* agents that will be moved as shown in the previous example when the consortium fires. For example, one enabled consortium is the one we just discussed which moves the agent $T1$ from place *seat* of the plane, to place *rf* of the airport agent that we just discussed. Another consortium is corresponding to agent $T2$, which in the configuration shown in the example is not enabled since $T2$ is not located in place *seat*.

One of the most important features of hypernets is that they have a straightforward expansion towards a behaviourally equivalent 1-safe nets. This expansion not only gives hypernets a precise semantics in terms of a well known Petri nets basic model, but also guarantees the possibility to reinterpret on hypernet all the analysis techniques developed for the basic model. The 1-safe net is built in the following way:

- For each agent A, and for each place p in the hypernet a place named $\langle A, p\rangle$ is added in the corresponding 1-safe net. A token in this place means that A is located in p,
- For each consortium Γ in the hypernet a transition named t_Γ is added in the 1-safe net,
- An arc is added from a place $\langle A, p\rangle$, to a transition t_Γ if A is a passive agent in Γ, and p is the input place from which the agent A comes.
- An arc is added from a transition t_Γ, to a place $\langle A, p\rangle$ if A is a passive agent in Γ, and p is the output place where the agent A is going to.

Finally, a place $\langle A, p\rangle$ of the 1-safe net is marked if in the initial configuration of the hypernet agent A is located in place p.

For example, the *one seater plane* case of study we just discussed is translated in the 1-safe net shown in Figure 4. Plane $P1$ can be in places lg, rf, bg in the hypernet, thus the 1-safe net contains places $\langle P1, lg\rangle$, $\langle P1, lg\rangle$,$\langle P1, lg\rangle$. The same must be done for traveler agents, and for the CHK check agent. Since transition *deplane* in the hypernet has two firing-modes, in the 1-safe net two transitions which models each of the firing modes of *deplane* are added (for simplicity both called *deplane*). The same has been done for transition *board*. The firing of a transition in the 1-safe net exactly models what happens when a consortium fires in the hypernet.

As already mentioned, it can be demonstrated that this net is 1-safe, and has a reachability graph isomorphic to the one of the corresponding hypernet. Details, formal definitions, and proofs discussed can be found here for hypernets in [3], and in [9] for the generalization version.

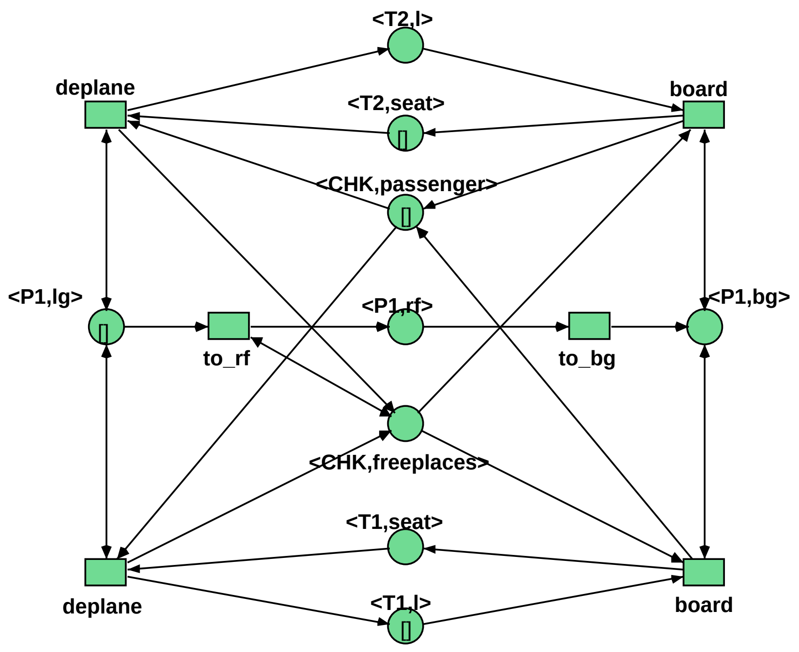

Fig. 4. The expansion toward 1-safe net of the hypernet in Figure 2 and Figure 3

3 Restricting Reference Nets to Hypernet

The main motivation for using high level nets is that, given a system, it is possible to obtain a model of the system with an high level net which is smaller compared to the model obtained using basic Petri nets. However, if you are not careful, the increase of the modelling power decreases the decision power of the model. For example, in [6] it was shown that, even considering a simple subclass of reference nets with one system net, and several references to an object net, the reachability problem becomes undecidable.

It is in this perspective that the implementation of the hypernet formalism as a plugin of the RENEW tool has been made. Restricting reference net is probably the most intuitive way to use verification techniques in RENEW. In particular, the use of a nets-within-nets formalism like hypernets as a restriction permits the use of the nets-within-nets paradigm, which is probably the most intresting feature in RENEW. The original contribute of the paper is to show how this plugin allows the use of verification techniques, like invariants and CTL model checking, to check properties of systems which are suitable to be modeled with the the nets-within-nets paradigm.

4 The Hypernet Plugin

From a technical point of view the implementation of a new formalism in RENEW is done using a plugin mechanism. The most important method contained in the classes implementing the plugin is a *compile* method which takes as input a *shadow* net, a set of Java objects containing all the information about the net the user has drawn in the graphical editor of RENEW, and transform it in a set of Java objects used by the simulator engine to simulate the net. This compile method is responsible for checking that the net drawn by the user is an actual hypernet in our case. In particular, in order to be able to use RENEW as a hypernet simulator, the arc and transition inscriptions used in the modeling process must be restricted in such a way that the drawn net is a hypernet. Therefore the restrictions applied in the plugin are the following:

– Inscriptions (tokens) inside places can only be in the following forms: *identifier* or *identifier:netType*. In the first case the identifier represent the name of an empty net, and will be treated by the simulator engine as an black token; in the second case a new instance of the net *netType* will be created and placed inside the place.
– Inscriptions on arcs are restricted to single variables only. Each arc must contain exactly one variable inscription.
– The inscriptions of input (output) arcs must not be duplicated. In this way it is possible to preserve the identity of nets: duplication of tokens is forbidden.
– Balancing of transition has to be checked, i.e.: the set of variable names used to inscribe input arcs must coincide with the set of variable names used to inscribe output arcs.
– Communication places are deleted, and are simulated by means of synchronous channels. These channels are counted when checking transition balance.

For example, the airport agent shown in Figure 2 can be drawn as a hypernet in RENEW using the net shown in Figure 5. The traveler empty tokens are place inscriptions $T1$ and $T2$, and the plane net instance is created by the $P1 : place$ inscription. Each transition is balanced. For example transition *deplane* in the airport has a bidirectional arc labelled *pl*, and an output arc labelled *pa* for which there is a correspondant downling, namely $pl : deplane(pa)$. Each communication place is deleted, and it is replaced with a synchronous channel. *Land* and *takeoff* transitions are equipped with two uplink because they were connected to two up-communication places. *Deplane* and *board* transitions contain two downlinks because they were connected to down-communicating places. The module name used to label communicating places is used to retrieve the variable name used in the downlink.

The $P1$ agent of Figure 3 is drawn in the hypernet plugin of RENEW with the net in Figure 6. Again, up-communication places are replaced by channels, and transition *to_ rf* must synchronise with the corresponding transition in the airport agent.

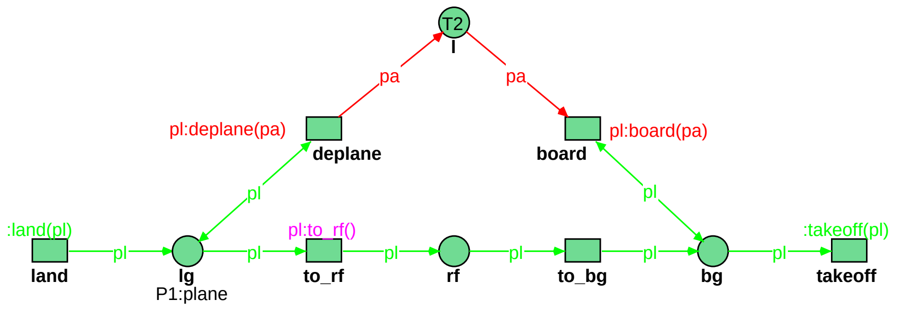

Fig. 5. The airport agent drawn with the hypernet plugin of RENEW

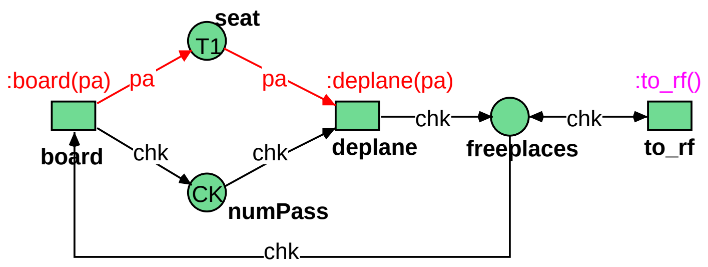

Fig. 6. The plane agent drawn with the hypernet plugin of RENEW

As we already mentioned, thanks to the expansion to 1-safe nets it is possible to use verification techniques defined for this class of net to analyse system modelled with a hypernet. Two of the most useful techniques are invariants analysis, and model checking. We explored two possibilities of using them in the plugin we implemented: internal implementation in RENEW, or exporting the 1-safe net in a format understandable by other tools. Since implementing these analysis techniques in an efficient way is a difficult task (some tools are very elaborated, and have been implemented over several years), and since very efficient open source tools are available for free, we decided to use external tools to implement invariant analysis, and model checking of a hypernet.

In the following sections we will show how the extensions and incorporation can be used in a practical example.

5 Example

The invariant analysis, and the model checking extensions we implemented in RENEW can be used to prove properties of a system. We have chosen the external tools LoLA (see `http://www2.informatik.hu-berlin.de/top/lola/lola.html`) and INA (see `http://www2.informatik.hu-berlin.de/~starke/`

ina.html) for analysing purposes. Starting from the hypernet example of Section 2.2, we will prove using invariants that there is never more than one passenger on the plane, and we will prove using the model checker that a plane never refuels if there are a passenger on board.

By running the invariant analysis we get the following invariants:

T2@l	T2@seat	CHK@pass	P1@lg	P1@rf	P1@bg	CHK@freepl	T1@seat	T1@l
0	0	0	1	1	1	0	0	0
1	1	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0	0
0	0	0	0	0	0	0	1	1
1	0	1	0	0	0	0	0	1
0	1	0	0	0	0	1	1	0

The first four invariants are those which guarantee the truth of "law of conservation of agents", achieved thanks to the state machine decomposition in the formalism. For each agent there is a corresponding invariant indicating the places in which that agent can be located. Since the places of each invariant contains only one token in the initial marking, it is mathematically proved that each agent can be only in certain places: the places which are of the same sort of the agent itself. Moreover, these four invariants can also be used to prove that the net is 1-safe: they cover all places of the net, and contain only one token in the initial marking.

The fifth invariant is $\{\langle T2, l\rangle, \langle CHK, numPass\rangle, \langle T1, l\rangle\}$ and contains two tokens in the initial marking. Together with the second and the fourth invariants it can be used to prove that if the place $\langle CHK, numPass\rangle$ is marked then one of the two passenger is seated on the plane. The place is not marked only if both passenger are in the airport.

The sixth invariant is the counterpart of the fifth, and states that only one of the following places can be marked: $\{\langle T2, seat\rangle, \langle CHK, freeplaces\rangle, \langle T1, seat\rangle\}$. The information is clear: only one passenger can be in the *seat* place of the plane. If none of them is in the plane $\langle CHK, freeplaces\rangle$ is marked.

In Figure 7 a screenshot of Renew after the computation of invariants is shown.

While invariants analysis can be launched, and the computed invariants can be analysed to extract information about the system, in order to analyze the system using model checking a formula specified in a temporal logic is needed. Since we choose LoLA, which is a CTL model checker, we need to specify the property we want to verify using this logic. For example, checking the property "if the plane is located in the place representing the refueling station then no passenger is on board" can be done by entering as input of the Renew plugin we implemented the following CTL formula:

$$\begin{array}{l} ALLPATH\ ALWAYS \\ NOT\ ((T1.seat = 1\ AND\ P1.rf = 1)\ OR\ (T2.seat = 1\ AND\ P1.rf = 1)) \end{array}$$

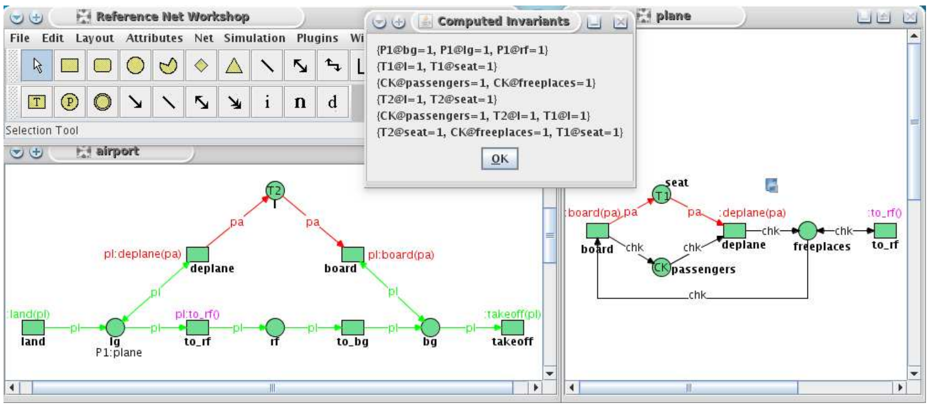
```
{P1@bg=1, P1@lg=1, P1@rf=1}
{T1@l=1, T1@seat=1}
{CK@passengers=1, CK@freeplaces=1}
{T2@l=1, T2@seat=1}
{CK@passengers=1, T2@l=1, T1@l=1}
{T2@seat=1, CK@freeplaces=1, T1@seat=1}
```

Fig. 7. A screenshot of the invariants computed inside RENEW

The formula checks that in every reachable state ($ALLPATH\ ALWAYS$) the situation in which both placed $\langle T1, seat\rangle$ and $\langle P1, rf\rangle$ are marked never occurs (and the same for places $\langle T2, seat\rangle$ and $\langle P1, rf\rangle$). The analysis performed confirms that the truth value of the formula is *true*, which is enough to guarantee that the property is true for the system.

As it can be seen in this simple example, the advantage of using model checking is that it is possible to express, and consequently to verify, more properties compared to invariant analysis. In our example, the information that a plane never refuels if a passenger is on board is not present in the computed invariants, but can be verified using the model checking. However, the drawback is that it is necessary to explore the whole state space of the system in order to verify a property. Invariants are computed on the static structure of the net, which is usually exponentially smaller compared to the state space of the system. In general, in real huge application both the techniques are useful: invariants give a quick overview of some properties of the system, model checking take more time and it can be used to verify specific properties of the system.

6 Conclusion

In this paper we discussed the verification of high-level Petri nets which use the nets within nets paradigm, with particular attention to the reference nets and the hypernets formalisms. We examined them, and we showed how to transform a subset of reference nets into hypernets, which in turn can be transformed into 1-safe nets. We then proceeded to describe the hypernet plugin created for RENEW in the course of our work. With the help of this plugin and external tools we can analyse the transformed low-level nets, and in this way verify properties of the high-level net.

The contributions, and the results of this paper are the implementation of a plugin for RENEW with which it is possible to draw of a hypernet, to compute its invariants, and to model check it. With this approach it is now possible to verify properties of systems modelled with net within nets oriented formalisms, such as reference nets and hypernets.

The results of this paper will make it possible to automatically analyse a hypernet, instead of first transforming it by hand, and then analysing the equivalent low-level nets. This will make the verification simpler and more user-friendly, which in turn will make it easier for software engineers to use these techniques in practical use cases. We plan to use these approaches to verify the model of an actually adopted Grid tool for High Energy Physics data analysis, and in the context of the HEROLD project. Future work will also focus on extending the possibilities of the verification, automating the process as far as possible and extending the toolset to other high-level Petri nets formalisms. The flexibility and adaptability of the Renew tool will be a large asset in this endeavour. Finally, the definitions of analysis techniques directly on the high level model, without the need of converting it to a low-level one, is a subject for future investigations, because it will avoid the conversion to low-level nets, which is an expensive operations in term of computational resources.

References

1. M Bednarczyk, L Bernardinello, W Pawłowski, and L Pomello. Modelling and analysing systems of agents by agent-aware transition systems. In F. Fogelman-Soulie, editor, *Mining Massive Data Sets for Security: Advances in Data Mining, Search, Social Networks and Text Mining, and their Applications to Security*, volume 19, pages 103–112. IOS Press, 2008.
2. Marek A. Bednarczyk, Luca Bernardinello, Wiesław Pawłowski, and Lucia Pomello. Modelling mobility with Petri Hypernets. In *Recent Trends in Algebraic Development Techniques*, volume 3423/2005 of *Lecture Notes in Computer Science*, pages 28–44. Springer Berlin / Heidelberg, 2005.
3. Marek A. Bednarczyk, Luca Bernardinello, Wiesław Pawłowski, and Lucia Pomello. From Petri hypernets to 1-safe nets. In *Proceedings of the Fourth International Workshop on Modelling of Objects, Components and Agents, MOCA'06, Bericht 272, FBI-HH-B-272/06, 2006*, pages 23–43, June 2006.
4. Luca Bernardinello, Nicola Bonzanni, Marco Mascheroni, and Lucia Pomello. Modeling symport/antiport p systems with a class of hierarchical Petri nets. In *Membrane Computing*, volume Volume 4860/2007 of *Lecture Notes in Computer Science*, pages 124–137. Springer Berlin / Heidelberg, 2007.
5. Soren Christensen and Niels Damgaard Hansen. Coloured petri nets extended with channels for synchronous communication. *Lecture Notes in Computer Science*, 815/1994:159–178, 1994. Application and Theory of Petri Nets 1994.
6. Michael Köhler and Heiko Rölke. Properties of object Petri nets. In Jordi Cortadella and Wolfgang Reisig, editors, *ICATPN*, volume 3099 of *Lecture Notes in Computer Science*, pages 278–297. Springer, 2004.
7. Olaf Kummer. *Referenznetze.* Logos Verlag, Berlin, 2002.
8. Olaf Kummer, Frank Wienberg, Michael Duvigneau, Jörn Schumacher, Michael Köhler, Daniel Moldt, Heiko Rölke, and Rüdiger Valk. An extensible editor and simulation engine for Petri nets: Renew. In J. Cortadella and W. Reisig, editors,

International Conference on Application and Theory of Petri Nets 2004, volume 3099 of *Lecture Notes in Computer Science*, pages 484 – 493. Springer-Verlag, 2004.

9. Marco Mascheroni. Generalized hypernets and their semantics. In *Proceedings of the Fith International Workshop on Modelling of Objects, Components and Agents, MOCA'09, Bericht 290, 2009*, pages 87–106, September 2009.
10. Einar Smith. Principles of high-level net theory. In *Lectures on Petri Nets I: Basic Models*, volume Volume 1491/1998 of *Lecture Notes in Computer Science*, pages 174–210. Springer Berlin / Heidelberg, 1998.
11. Rüdiger Valk. Object Petri nets: Using the nets-within-nets paradigm. In Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Advanced Course on Petri Nets 2003*, volume 3098 of *Lecture Notes in Computer Science*, pages 819–848. Springer-Verlag, 2003.