



Distributed Petri nets for model-driven verifiable robotic applications in ROS

Sebastian Ebert^{1,3} · Johannes Mey^{2,3} · René Schöne^{2,3} · Sebastian Götz³ · Uwe Aßmann^{1,2,3}

Received: 17 January 2024 / Accepted: 27 May 2024
© The Author(s) 2024

Abstract

Verifying industrial robotic systems is a complex task because those systems are distributed and solely defined by their implementation instead of models of the system to be verified. Some technologies mitigate parts of this problem, e.g., robotic middleware such as the Robotic Operating System (ROS) or concrete solutions such as automata-based specification of robot behavior. However, they all lack the required modeling depth to describe the structure, behavior, and communication of the system. We introduce an improved version of our previous model-driven approach based on Petri nets, integrating these three aspects of ROS-based systems. Using a formal modeling language enables verification of the described system and the generation of complete system parts in the form of ROS nodes. This reduces testing effort because the specification of component workflows and interfaces remains formally proven, while only changed implementations have to be revalidated. We extended our previous approach with novel model transformations, which considerably improved our approach's performance and memory requirements. We evaluate our approach in a case study involving multiple industrial robotic arms and show that the structure of and communication between ROS nodes can be described and verified.

Keywords Petri nets · Model-driven engineering · Robotic software engineering · Robot operating system

1 Introduction

Developing and extending robotic software in industrial or end-user applications, especially with human interaction, is an intensive task. Such software comprises physically distributed communicating components, which need to adapt

to varying environments and tasks. Hence, these robotic systems are complex and safety-critical and require a huge effort to test, develop, and extend them.

The creation and evolution of robotic systems are often realized ad hoc, resulting in unstructured systems defined entirely by their implementation[1], complicating validation and preventing verification. On the contrary, constructing these systems upon formal models allows the automatic addressing of properties such as safety and reliability before the actual implementation, during and after runtime. An instance of this is defining a safety model and examining its state transitions. The de-facto standard robotic middleware, the Robot Operating System (ROS)[2], also does not provide a formal foundation for creating and verifying models to guarantee the global and concurrent behavior of distributed applications. Furthermore, the verification should reflect the influence of other applications and models to check mission-critical reactions to changing environments. Existing approaches building on top of ROS either enable the modeling of infrastructural aspects of ROS, verify parts of ROS applications, or deal with the influence of external applications [3–22]. Our approach, Distributed Petri Nets for ROS

✉ Sebastian Ebert
sebastian.ebert@tu-dresden.de

Johannes Mey
johannes.mey@tu-dresden.de

René Schöne
rene.schoene@tu-dresden.de

Sebastian Götz
sebastian.goetz1@tu-dresden.de

Uwe Aßmann
uwe.assmann@tu-dresden.de

¹ Centre for Tactile Internet with Human-in-the-Loop (CeTI), Dresden, Germany

² 6G-life, Dresden, Germany

³ Technische Universität Dresden, Chair of Software Technology, Dresden, Germany

(DiNeROS), aims to address all three aspects by using Petri nets to realize a model-driven toolchain for ROS applications generating stubs as an integration target for user-defined implementations.

Petri nets were chosen for this work because they provide advantageous features for the field of robotics compared to other formalisms. Firstly, they allow modeling concurrency, a common characteristic in robotics where multiple actions occur simultaneously on possibly distributed devices. Secondly, Petri nets are nondeterministic, which matches the aspect that robotic environments contain uncertainties, such as changing environmental conditions and unpredictable events. Thirdly, Petri nets offer high expressiveness, allowing the modeling of more system states within a smaller structure compared to, for example, automata. Robotic applications typically provide a large state space due to interactions with their environment. Finally, Petri nets offer a graphical representation allowing to visualize the behavior of robotic systems, aiding understanding interactions and dependencies.

Communication of ROS applications quickly becomes complex, even for small applications. For example, changing communication channel capacities generates side effects on other indirectly connected components. Formally describing the coordination of distributed actions, e.g., multiple moving robots working in the same environment, enables their verification and improves extensibility compared to a hand-crafted implementation. Therefore, the models need to include not only behavior but also communication and structural aspects of ROS and provide a formal underpinning to develop and verify ROS applications efficiently, leading to the first research question **RQ1**: *How to enable the development of verified ROS applications using Petri nets as formal underpinning?* ROS-based applications may communicate with external components, such as sensor interfaces or proprietary software components, whose interfaces are often not formally modeled, leading to **RQ2**: *How to enable the integration of other formalisms and non-formal components?* We use Petri nets as a formal modeling language, leveraging their non-determinism to control the application's behavior. This leads to **RQ3**: *How to use Petri nets as runtime models [23] in ROS applications?*

This paper is an extended version of a previously published workshop paper [24]. We extended the original approach with novel model transformations on signaling and ROS topics, which are considerably more performant and less memory-demanding. Moreover, we provide a much more detailed description of the analysis features provided by our approach. This includes firstly new state space analysis algorithms reasoning about communication behavior, dead application parts and effects of external components. Secondly, we present a new tool to visualize traces at modeling time and runtime.

We evaluate our approach using a case study, where multiple industrial robotic arms are coordinated to sort objects within a shared workspace based on their properties. Using our model-driven approach, we can verify the resulting ROS application and identify incorrect system constructions, such as an unrealizable condition.

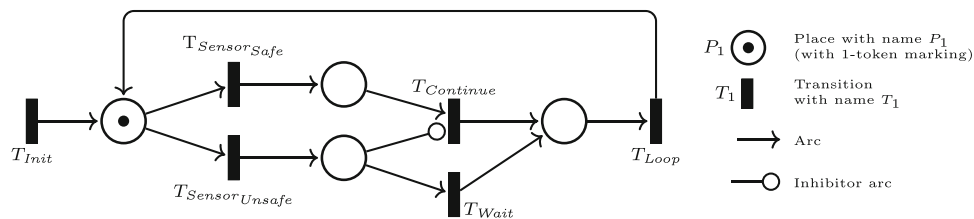
This paper is structured according to the following outline: Sect. 2 introduces the Robot Operating System and the Petri net formalism, Sect. 4 describes the case study, followed by Sect. 5 describing the model-driven development chain, as well as the integrated models and their transformations. Section 6 models the case study, and Sect. 7 analyzes the constructed model. Section 8 treats the performance aspects of analyzing the model, while Sect. 9 shows how to debug the model and the resulting application using traces. Finally, Sect. 10 discusses the feasibility of our approach, and Sect. 11 concludes the paper.

2 Background and terminology

Our approach is based on **Petri nets** as defined in ISO/IEC 15909, where a *Petri net* (PN) is a directed bipartite graph $PN = \langle P, T, F, R_P, R_T, P_G \rangle$ consisting of a set of *places* P , depicted with circles, and *transitions* T , depicted with bars, connected by *arcs* defined as the incidence flow relation $F \subseteq (P \times T) \cup (T \times P)$, where $S_A(a)$ and $T_A(a)$ are source and target of an arc $a \in F$. Places, drawn as circles, contain a discrete number of tokens represented by dots or their number in places. The distribution of tokens within the places is called a *marking* $M_i = [m_1(i), \dots, m_n(i)]$ for a Petri net at time i with n places. A transition, represented by a bar is enabled and thus can fire if there is at least one token in each input place, hence $\forall M_i(s) \geq 1$. If a transition, fires, tokens from each input-place of the transition are destroyed and newly created inside every output-place of the firing transition, creating a new marking M_{i+1} [25]. The number of tokens to be destroyed and created depends on the weight of the respective arc between place and transition. An essential characteristic of Petri nets is that transitions fire non-deterministically, i.e., if multiple transitions can fire, no precedence is given to any of them. An *inhibitor arc*, marked by a circle at the arc target, enables a transition if the source place is empty [26]. In the following, we refer to all objects in a Petri net as $O = T \cup P \cup R_T \cup R_P \cup P_G$, which is the set of all non-arc elements and to a labeling assigning objects to pages as a function $l : O \rightarrow P_G \cup \Theta, o \rightarrow p_G \cup \Theta$, where Θ defines that no page is assigned.

Structuring Petri nets is done with a set of *pages* P_G , *reference transitions* R_T , and *reference places* R_P . A page may contain other pages, defining a hierarchy of sub-pages, where arcs must connect transitions, places, and references

Fig. 1 A nondeterministic safety controller modeled by a Petri net



on them within the same page, defining an assignment for page i as $S_i = \{o \mid o \in O \wedge l(o) = i\}$. Thus, pages do not add semantics but facilitate the partitioning of Petri nets. Connecting pages is done by selecting a representative place or transition in one page and integrating it into the other pages as respective references, which are attached as a property to referencing elements. Reference places or transitions may be connected with other places or transitions within the same page by arcs, but cyclic references are not allowed [27].

The *Petri Net Markup Language* (PNML) provides an XML-based interchange format for Petri nets. PNML supports Petri nets with references and high-level Petri nets [28]. Moreover, PNML is modular and provides generic extension points for additional information for each element. *TINA* [29] is a toolbox providing the abilities for editing and analyzing Petri nets.

Petri nets have special properties, making them a great candidate for modeling robotic systems. First, their non-deterministic [25] nature reflects the non-determinism of distributed robotic environments and enables modeling synchronous and asynchronous control. An example of this, a simple safety model, is shown in Fig. 1, where T_{Safe} and T_{Unsafe} represent the results of a sensor detecting humans within a robots range. Depending on the fired transition either the currently performed task is continued ($T_{Continue}$) or the robot needs to wait (T_{Wait}). Second, the state of a PN is represented by its marking, i.e., the distribution of tokens to places depicted by dots within places. This enables modeling concurrent control flow, an inherent characteristic of robotic applications.

Third, PNs provide various analyzable properties, enabling automated verification over concurrent systems. Most importantly, building state spaces based on a PN specification to reason about the reachability of system states is possible. Furthermore, it is possible to analyze the boundedness, liveness, and deadlocks of a Petri nets marking [25, 26].

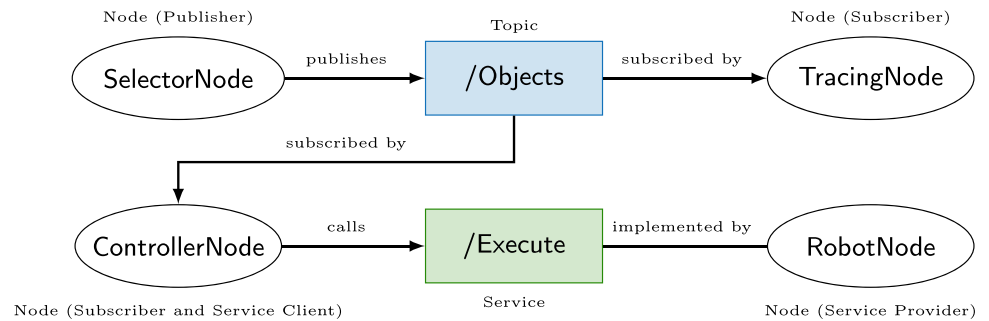
The **Robot Operating System** (ROS) is an open-source middleware providing tools and libraries facilitating the development of robotic applications. ROS enables the creation of components structured within packages and nodes, providing control of robotic hardware and pure software functionalities. Nodes themselves are communicating based on topics and services. A ROS-based process called *node* is

registered at the ROS *master*. Libraries and nodes are grouped within *packages*, enabling reuse and loose coupling [2]. An example of such a package is displayed in Fig. 2, where nodes are depicted as ellipses, topics as blue boxes, and services as green boxes. Within the example, objects are selected in an according node and sent to a node logging the selected objects and a controller executing actions on selected objects by accessing a robot controlling node.

ROS provides a structured communication layer, letting nodes communicate based on messages within topics and services. *Topics*, such as the */Objects*-topic, are uniquely named and based on the publish-subscribe pattern. Each publisher has an outgoing message queue of a specific size, where new messages get lost when it is exceeded, so selections of objects within the example may get lost. For subscribers, dispatchers and callbacks exist. When a node has subscribed to a topic, it runs a dispatcher thread for that topic, collecting messages within a queue with fixed capacity and distributing messages to callbacks listening to the topic within the node, which again has its own bounded queue. Types of messages are user-defined and vary between topics. Synchronous communication is enabled by *services*, following the request-response pattern, where each client can send at most one request in parallel and blocks while waiting for the response, which means the call to the */Execute*-service within the example is synchronous. The server side is controlled by a thread pool, enabling handling requests in parallel. On arrival within a server instance, the request is processed, creating a single result returned to the calling client, who may call other services [2]. These underlying mechanisms vary between client library implementations. The core of ROS is usable independently of any programming language. Client libraries are provided for languages like C++, Java, and Python. *RosJava*, developed at Google, provides basic ROS functionalities such as topics, services, and nodes within the Java environment [30]. Our focus is the common subset of the Java and C++ implementation.

3 State of the art

Software development based on formal models for robotic applications is already applied by various approaches, focusing on ROS aspects, application models, or both. In this

Fig. 2 Visualization of a simple ROS package**Table 1** Related ROS approaches, with full support denoted with ■, partial support with ⊞, and no support with □

ROS-based approach	ROS structure	ROS communication	External components	Model-driven development	Modeling approach	Formal approach
Lesire [3]	□	⊞	■	■	Petri nets	■
Dondrup [4]	□	⊞	□	■	Petri nets	■
Pelletier [5]	□	□	■	■	Petri nets	■
dos Santos [6]	□	□	■	■	Petri nets	■
Figat [7]	□	□	■	■	Petri nets	■
Dal Zilio [8]	□	□	■	■	Petri nets	■
Halder [9]	■	⊞	□	□	Automata	■
Wang [10]	■	⊞	□	■	Automata	■
Cheng [11]	⊞	□	■	■	GSN	□
Kortik [12]	⊞	⊞	■	□	Linear Logic	■
Zander [13]	⊞	□	□	■	Ontologies	■
Estévez [14]	⊞	⊞	⊞	■	RDIS	□
Chaudhuri [15]	□	□	⊞	■	CNL	□
Kilgo [16]	⊞	□	⊞	■	UML	□
Beaulieu [17]	□	□	⊞	■	UML	□
Brugali [18]	⊞	■	⊞	■	DSL	□
El Baccouri [19]	⊞	⊞	⊞	■	DSL	□
Ramaswamy [20]	□	□	⊞	■	DSL	■
Baumgartl [21]	□	□	⊞	■	DSL	□
Heinzemann [22]	□	■	⊞	■	DSL	■
DiNeROS (<i>this work</i>)	■	■	■	■	Petri nets	■

section, we compare existing approaches and our approach Distributed Petri Nets for ROS (DiNeROS) w.r.t. our three modeling requirements (structure, communication, external components) and usage of (formal) Model-driven development (MDD). The analyzed approaches are compared in Table 1, where full support in terms of communication is about models specifying topics and services, while the structure is about nodes and packages. Modeling topics and services allows to reason about communication based behavior, by refining the application describing models. Modeling nodes and packages allows using the application's distribution within reasoning, increasing its expressiveness. External components treat formally defined (full support), programmatic (partial support) and no interfaces. MDD can be either

supported or not. The following section takes a closer look at ROS-based approaches.

3.1 Petri net based ROS approaches

ASPiC [3] introduces constrained Petri nets called control-flow Petri nets modeling robotic skills, parameterized by required resources (locks, inputs, outputs). The approach models ROS action interfaces abstractly, not including the underlying topics and services. Structural aspects are not described and the constructed models are used to generate state spaces and check well-formedness criteria. Dondrup et al. [4] handles interactions between robots and humans with interaction patterns modeled as Petri nets. Action interfaces are modeled even more abstractly than in [3], while

the actual verification is not described. Within SkiNet by Pelletier et al. [5], where the skill-based system architectures are transformed into Petri nets. These Petri nets model the state-machine-based behaviors of robotic skills used resources. Regarding model-checking, user-defined temporal logic specifications are evaluated at runtime using the TINA [29] tooling. Dos Santos et. al [6] introduces a software framework allowing the execution of a robot task, defined by Marked Petri nets, which e.g. extend places with capacities. The framework is built on top of ROS and is capable of real-time execution of hierarchically structured tasks. Furthermore, the framework provides structural verification tooling for marked Petri nets, without considering any ROS concepts. Figat et. al [7] provides a framework based on hierarchically structured Petri nets allowing to model distributed robotic systems. A specification language is used to construct such systems. The languages instances are verified by transformed them into a six layered Petri net, where further analysis are not described. Additionally, controller code is generated based on these models. The framework is ROS-based, but the models do not contain ROS concepts in detail. Dal Zilio et. al [8] defines a domain-specific language based on an extended version of Time Petri nets. The approach describes how these formal models can be executed and how they provide the basis for monitoring and runtime verification, aided by the TINA tooling [29]. No ROS concepts are modeled.

All Petri net-based approaches introduce an MDD chain for ROS applications to generate executable code. However, no approach is available for extension or fully supports modeling structural and communication aspects of ROS.

3.2 Model based ROS approaches

Halder et al. [9] use timed automata and verify ROS applications based on real-time properties with Uppaal [31]. The approach enables modeling control flow concepts within ROS nodes and topic-based messaging but no formal interfaces. Uppaal allows forward analysis based on extrapolation, which is used in this work to vary combinations of parameter values of ROS nodes. Wang et al. [10] propose an MDD method based on timed automata, generating executable ROS code. Their approach enables checking safety requirements using the Uppaal query language to detect deadlocks. The expressiveness of the models is comparable to [9].

Cheng et al. [11] focuses on guaranteeing safety requirements during runtime adaptation. Goal Structuring Notation (GSN) models are used as a specification language for a central adaptation node. Only structural aspects of ROS are modeled. Kortik et al. [12] focus on verifying correctness based on linear logic, where the correct behavior is verified by analyzing the computational graphs consistency (idle publishers/subscribers, type consistencies). Structural

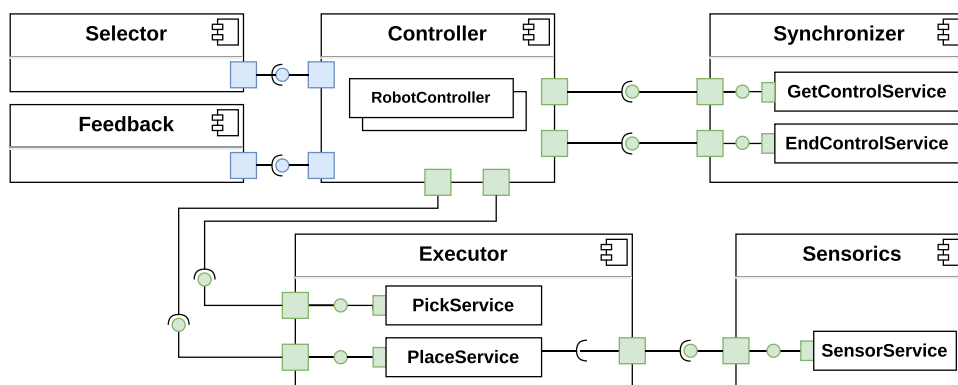
aspects (nodes) and communication aspects (topics) are modeled abstractly. The approach of Zander et al. [13] provides an MDD toolchain built upon ontological semantics, where developers create a local model including capabilities and interfaces of robotic software components. The modeling approach does not cover communication aspects. Verification and validation algorithms are demarcated as future work. Chaudhuri et al. [15] use a constrained natural language and an ontology-based knowledge representation to express robotic capabilities and workflows to generate ROS code. No aspects of ROS are modeled, while external components are included programmatically. Kilgo et al. [16] define a metamodel describing ROS applications, including topic interfaces combined with the Robot Device Interface Specification (RDIS) describing hardware interfaces. External components are only integrated when having RDIS interfaces. Beaulieu et al. [17] uses UML and Finite State Machines for multiple robotic application areas: localization, mapping, and swarming. It was implemented in ROS without modeling it or describing its verification.

Summarized, many approaches introduce an MDD chain. No approach is available for extension or supports modeling ROS structure respectively communication. Additionally, [11] and [12] provide formal interfaces for external applications.

3.3 DSL based ROS approaches

The ART²ool [14] provides a modeling mechanism for UML-based task models, where communication is abstractly described and transformed to ROS code and other middlewares. The code is the only interface for external models. HyperFlex [18] enables designing robotic software as a hierarchical composition of functional subsystems. Variation points for resources, algorithms, and control are modeled and resolved by a feature model. A control system varies these points. The approach models components and interfacing of nodes, services, and topics. In [19], parameterization and scenario-centralized modeling are used to evaluate different control settings and environmental contexts. Based on their DSL, code is generated mapping to ROS packages, nodes, and topics. Saferobots [20] divides software development into problem, solution, and operational space, whereby each space has a knowledge and application level. The modeled solution space is used to generate ROS code. Problem specific models can be simulated by an editor, providing a simple form of verification. Baumgartl et al. [21] use a DSL to specify tasks at a high abstraction level and encapsulate operations. The DSL is transformed into C++ code. ROS concepts are not part of the DSL. The vTSL DSL [22] enables specifying robotic tasks and verifying them against a set of safety and integrity constraints by transforming the vTSL to Promela models verified by the SPIN model checker [32]. Especially

Fig. 3 Architecture of our case study. Services are green and topics blue



the existence of deadlocks, temporal properties and satisfactions of assertions is checked. The approach focuses on topics and services.

In summary, all DSL-based approaches provide an MDD chain, while [18] and [22] model communication aspects of ROS. Furthermore only [22] is the formal approach of [18, 22], but provides only code-based integration of external applications. Again, no approach is available for extension.

3.4 Non ROS based approaches

Formal approaches are also used in non-ROS robotic applications. There, the majority of approaches use state-transition systems to describe system structure and application flow [33, 34]. Petri nets are used in modeling, where [35] proposes distributed, locally synchronous Petri nets connected by asynchronous channels and verified by provided tooling. Bera [36] integrates interaction patterns with communicating Petri nets within an architectural framework. Robotic Task Models [37] run Petri nets with a central coordinator providing robotic workflows constructed upon primitive tasks. Kotb et al. [38] enable distributed robotics to collaborate by modeling their abilities with Petri nets.

4 Running example

To illustrate our approach, let us introduce our case study: interactive, collaborative, and human-aware sorting of colored objects by two robots in a shared space, where the application parts are distributed among multiple ROS nodes. Sorting is a key application for industrial tasks.

Conceptually, the case study shown in Fig. 3 consists of six components, where each component is realized as a ROS node and implemented by its developer team. The *SELECTOR*-component represents the handling of user input, which is about creating selection tasks for objects to be sorted and implicitly the robot executing the sorting operation. The selections are sent to the *CONTROLLER*-component containing one robot controller for each robot, assigned to colors

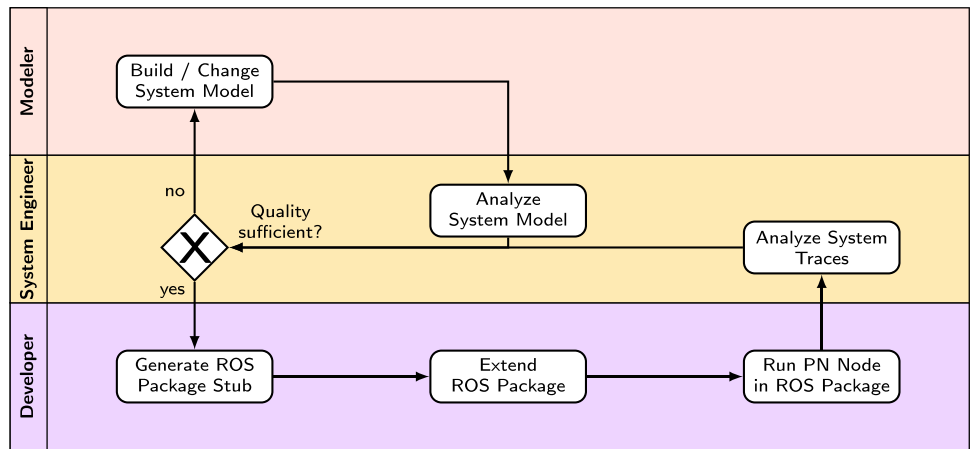
of objects by the connected topic. These robot controllers represent the central workflow that describes the steps to sort an object and coordinates the other components. The controller component is connected to the *SYNCHRONIZER* component, which acts as a semaphore for the shared workspace of the two robots. To control the robot, the controller calls pick and place services located within the executor component, developed by robotics experts. These services let the robot perform the trajectories for picking and placing but can also abort the execution based on a safety model handling human presence. These safety models obtain information about human presence based on the *SENSORICS* components service implemented by a sensor integration team. The *FEEDBACK* component receives the results of a sorting operation based on the subscription to the feedback topic and displays the received information within a user interface. For evaluation, the case study is subject to multiple variations. We change the number and types of objects and modify the control of the robots used for sorting, resulting in effects all over the distributed system.

5 Model-driven development with DiNeROS

Model-driven development enables the creation of applications based on models representing them. **Distributed Petri Nets for ROS (DiNeROS)** uses this concept and describes an MDD toolchain in combination with multiple extensions to the Petri net (PN) formalism supporting the construction of efficiently verifiable ROS applications by generating ROS package stubs from PN based specifications. These packages represent the border between verification and validation. This is because the stubs follow the PN-based specifications, while the actual actions, optionally attached to the model transitions, are implemented by developers extending them at the toolchain's end, which are subject to testing. The next sections describe the available MDD tooling,¹ the modeling approach, and how they interact.

¹ <https://dineros.pages.st.inf.tu-dresden.de>

Fig. 4 Workflow within the DiNeROS MDD



5.1 Overview

We differentiate three interacting roles in development as shown in Fig. 4. A *modeler*, expert in system design, is responsible for creating a Petri net based model defining and controlling the system, while a *developer*, expert in component construction, generates a ROS package based on this model and implements the concrete actions either executing the PN or being defined by the transitions of the PN. The *system engineer*, expert in formal methods, is firstly responsible for verifying the system defining model, and secondly analysing the behavior of DiNeROS applications, based on traces, and reporting detected issues to the modelers. Participants in the development process take exactly one of these roles, which are orthogonal aspects, allowing mutual control of their results, improving the software quality.

Essentially, DiNeROS covers three Petri net based models, which are embedded within a toolchain shown in Fig. 5: a System Model (SyM), a Runtime Model (RTM), and a Petri Net Model (PNM). Each model describes a different view within the MDD toolchain shown in Fig. 5. The SyM describes a distributed ROS application within a single model, containing communication channels (topics and services) as first-class citizen modeling elements. Additionally, signals are attached to transitions and combined to clauses, providing an externally accessible enabling mechanism. Thus, those clauses model all possible states of an input. An RTM is a PN describing exactly one ROS node and, therefore, only covers inputs and outputs of topic- and service-based communication. An PNM provides the same concepts as a SyM but is transformed into a PN, as defined in Sect. 2, modeling the internal mechanisms of ROS and enabling analyzability through existing model checkers.

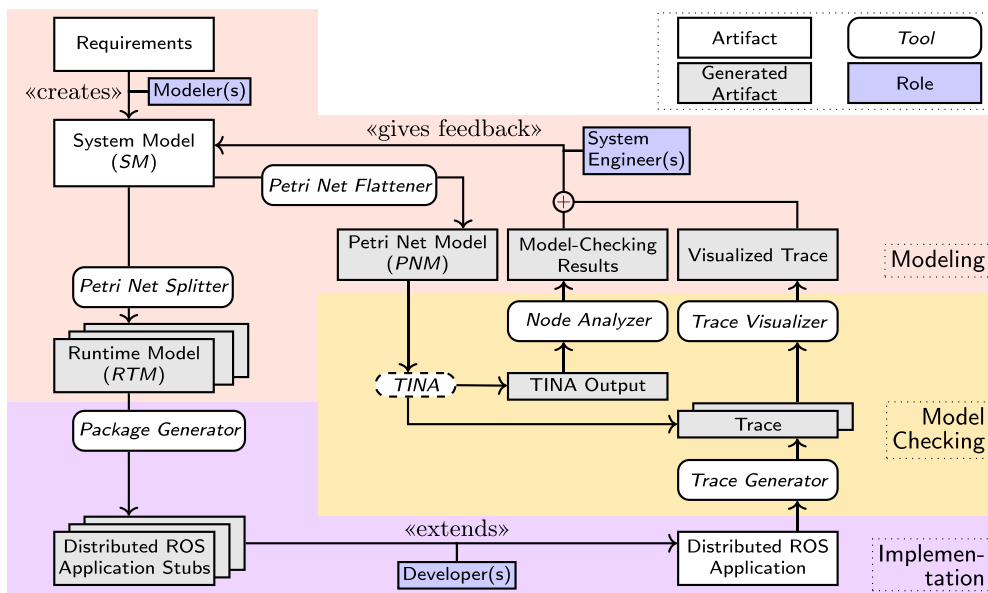


Fig. 5 MDD chain of DiNeROS

Additionally, we introduce the concept of *node nets*. One node net defines the model contained in exactly one ROS node. A node net itself is defined by an annotated page of a PN, and therefore contained in all three models. A node net is solely connected to other node nets through references from communication channels representing topics and services. In other words, all elements of a node net can only be connected to elements being part of the same node net or by incoming references from channels.

5.2 Model-driven development chain

Our MDD toolchain is visualized in Fig. 5, where all tools but TINA are newly developed based on JastAdd [39] and lip6 [40]. JastAdd is a compiler construction system based on attribute grammars, transforming the latter to Java classes. The process starts with the specification of a SyM based on PNML extended with the previously described features, where the extensions are domain-specific elements for topics, etc. This specification is manually performed by modelers based on given requirements. The chain is built on top of a runtime model-based environment, the *Petri Net Engine*, extending default PN execution in four ways: balloon tokens, advanced transition logic, handlers, and signals. *Balloon Tokens*, inspired by [41], replace the primitive tokens of PNs to enable data transfer between nodes and model parts. The data within these tokens is opaque concerning the execution semantics of PNs by requiring the same data format for all tokens. By doing so, we avoid the immense state space growth caused by heterogeneous tokens [42], but still allow data transfer. Balloon tokens are objects containing data fields, lists, and maps. Transitions have also been extended to handle balloon tokens and to attach developer-defined logic. If a transition in DiNeROS fires, it consumes one balloon token from each input place with possibly differing contents. Developers can register handlers at transitions to specify the processing of balloon tokens. The handlers attached to transitions own a priority, allowing not just to exchange them at runtime, but additionally to change their execution order based on. Handlers are not allowed to modify the marking or structure of a Petri net because this would break the execution semantics of Petri nets. Handlers provide a programmatic interface for the developers to write applications, extending the models they are attached to. They serve as interfaces to loosely couple code and models, enabling efficient testing of individual code contained in a handler without the need to re-test the whole application after changing it. Listing 1 shows an example of a handler attached to $T_{Feedback}$ in Fig. 14, 11 reporting the result of an operation.

The engine also automatically handles pushing received messages to input places of a receiving channel side. Vice versa, the engine provides interfacing for pushing messages from the sending side channel element to the receiving side.

Messages are received based on the specified names of topics and services, and the respective subscriber and server-side output places are automatically filled with a token containing the message content. For this, the engine integrates PNs into the lifecycle of ROS nodes as interpreted runtime models. Whenever the marking or signals change within a node net, and there are enabled transitions, the engine uses a function provided by the developer to select up to one to fire. Such a function is shown in Listing 2, where firing the transitions T_{Unsafe} and T_{Safe} is prioritized over the others depending on the input tokens content. Signals are connected to the environment based on channels connected to transitions, realized with RagConnect [43]. Moreover, the engine-integrated trace generator creates a trace at runtime containing fired transitions, signaling activity, and usages of communication channels.

The engine treats RQ3 using RTM models directly within ROS nodes as a runtime model controlling the application. RQ1 is partly treated by providing a systematic approach integrating formal modeling with PNs and writing application code connected to them. The message-based interface for input signals and clauses treats RQ2 as they enable integration of external applications through formally defined interfaces.

Listing 1 User-defined handler for $T_{Feedback}$ with priority 1

```

1  petrinet.registerHandler("T_Feedback", 1, (tokens) ->
2  {
3      SortingControllerToken result = tokens.get(0);
4      if (result.isPickPlaceSuccess()) {
5          print("Successfully sorted " + result.getObject());
6      }
7      } else {
8          print("Failed to sort " + result.getObject());
9          print(result.getErrorList());
10     }
11     return List.of(result); //last handler returns one
12     token
13 }
14 );

```

Listing 2 User-defined controller callback

```

1  @Override
2  FiringSelection onChange(Set<Transition> enabledTs) {
3      // check for prioritized enabled transitions
4      for (Transition t : enabledTs) {
5          if (t.getId().equals("T_Unsafe"))
6              if (t.getInputToken("P_SensorResponse")
7                  .get("humanDetected") == true)
8                  return new FiringSelectionSuccess(t);
9      }
10     for (Transition t : enabledTs) {
11         if (t.getId().equals("T_Safe"))
12             if (t.getInputToken("P_SensorResponse")
13                 .get("humanDetected") == false)
14                 return new FiringSelectionSuccess(t);
15     }
16     // if there is none, fire any other enabled
17     transition
18     return super.onChange(enabledTs);
19 }

```

The system model is used as input for the *Petri Net Flattener* tool to a Petri net model, removing all special language features of the SyM by applying the transformations described in Sect. 5.5. Finally, the flattened model acts as input for

the TINA tooling landscape, whose output is analyzed by the system engineer. If the model fails to satisfy the initial requirements, the SyM is revised, establishing a feedback loop by restarting from the DiNeROS PNML. This loop is run until the quality of the model meets the requirements.

The *Trace Visualizer* assists the system engineer in analyzing execution paths contained in the results of TINA and the engines runtime traces. For this purpose, a graphical representation is generated as depicted in Sect. 9, which is continuously updated at runtime or can be used at modeling time to visualize a selected TINA-generated trace.

The *Net Analyzer* provides algorithms for static model-checking, which further analyze the results provided by the TINA-provided tools. These algorithms are presented in Sect. 7 and provide further insights on communication and signal defects as well as unused application parts.

The *Petri Net Splitter* transforms the SyM to RTMs. It splits the channels of both services and topics on the sending and receiving sides and assigns the newly created elements to node nets based on their inputs on the sending side and outputs on the receiving side. In the second step, these models become a new RTM. Pages spanning multiple node nets are copied to each PN with elements located within such a page. Finally, each net is exported as a separate PNML file defining the runtime models.

The *Petri Net Package Generator* takes the various generated RTMs and generates ROS nodes based on the description having the RTMs as runtime models. The generated nodes are automatically started up and configured. The generator is integrated with the runtime engine to enable this, providing the execution logic interfacing with the developer. The generated package contains extension points for message handling, logic-to-transition bindings, and business logic. Furthermore, these extensions, if needed, are manually added by the developer, based on callbacks and handlers. Within the final stage of the chain, developers extend the generated ROS nodes corresponding to the respective models, resulting in a distributed deployable verified application. Whenever a ROS package is newly generated because the SyM model changed, all base classes and files are also newly generated. Thus, for now, developers must separate their handwritten code and call it from the generated classes to keep them. In summary, the MDD toolchain provides the necessary features to treat RQ1 by using PNs as a formal foundation to generate ROS packages. The three models (SyM, RTM, and PNM) treat RQ1 by providing a formal underpinning mapped to PNs with inhibitor arcs and treating RQ2 by providing signals as integration interfaces for other formalisms and components.

5.3 System model (SyM)

The SyM, the first artifact shown in Fig. 5, of a distributed ROS application is a single model and introduces the con-

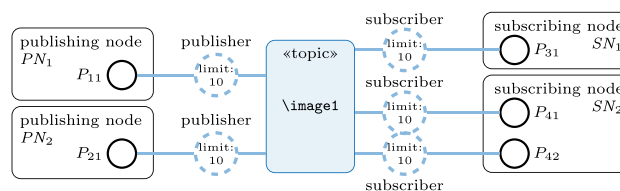


Fig. 6 Example topic model within a System Model

cepts of services, topics, signals and node nets to PNs. These concepts are later transformed to a PN in Sect. 5.5. Introducing these concepts as new modeling elements instead of using places and transitions simplifies the complexity of the model and improves its readability and usability.

5.3.1 Topics

A topic is represented by a new named element, the *topic channel*, shown as a blue box in Fig. 6, providing references connected to places working as input- and output ports. Each subscribing node has one subscriber for the topic, while the directly subscriber-connected places represent entry points of callbacks registered on subscribers. Each callback entry modeling place is part of a callback-modeling page within the subscribing node net. In the example, SN_2 has two callbacks on a subscriber to the `image1` topic. A topic channel submits tokens to its outputs when at least one of its connected input places has a token. It removes exactly one token from one input place—the message to be transported—and places it as a token in each subscriber-side output place. For each unique topic, there is exactly one responsible topic channel, preventing ambiguity. Each topic channel assigns capacity limits for each publisher and subscriber callback, as publishers and subscribers are also parameterized in this way within ROS to define the capacities of sender and receiver buffers.

5.3.2 Services

Synchronous and bidirectional communication is provided by services, which have a pair of *service channels*, shown as green boxes in Fig. 7. A service channel models both the request and the response part of service communication. Again, references are provided, connecting the input and output places of clients and servers, represented as input and output ports. A service channel element has per-client sub-elements representing the individual connections from each client to the according server. Each server element, defined within its `<<server>>` page, e.g., on the right side of Fig. 7, is a prototype, which means that for each client, there will be a runtime instance of it, executing requests independently. A client-wise channel is enabled if the *request* port of the channel contains a token, which was submitted to the connected server as shown on the left of Fig. 7. After this, and when a

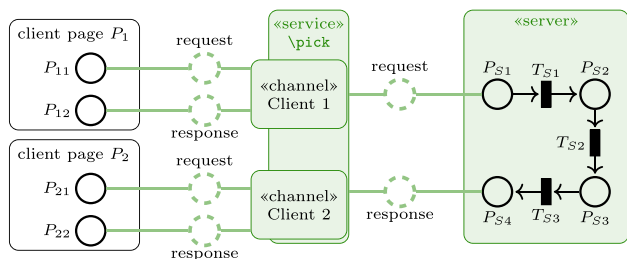


Fig. 7 Example service model within a System Model

token is placed accessible for the channel via the server-side response place (for example, P_{S4} in Fig. 7), it is sent back to the calling client based on the client-side response port. The calling direction is blocked as long as a client has not received a response from the previous request.

5.3.3 Signals

Integrating external components, which are not based on PNs, is realized through signals. Formally describing signals enables the integration of all possible signal configurations into model-checking. Binary valued and arc indicated *input signals* (IS) are an extension of the enabling mechanism of transitions shown in the upper part of Fig. 11. Input signals can be linked by the logical operators \wedge or \vee , and inverted by \neg , within a conjunctive normal form, enabling arbitrarily complex logical named expressions, referred to as *input signal clause* (IC). Input signals can be applied to multiple transitions, but only within the same node, so topics and services remain the only interaction mechanism for nodes. Multiple occurrences of input signals cause interactions. Modeling ICs with Petri nets makes these interactions analyzeable. Input signals extract a formally sound subset, extending existing works such as [44, 45].

5.3.4 Metamodel of the system model

Each System Model is based in fixed metamodel where an extract of it as an *AST*-model is shown in Listing 3, extending the ISO defined metamodel.

First, on lines 1 and 2, we extend transitions and places by subclasses, adding additional information. A *TransitionInfo* defined on line 7 has three subclasses which are integrating signal clauses to transitions (line 14), or integrate service- and topic channels. The connection of these channels to the places of publishers, subscribers, clients, and services is then defined in ports starting at line 16. Therefore, we have different port types, where topic ports provide capacities and service channel ports define the connection between requesting and responding places. Hence, a topic always provides multiple subscribers and publishers, while a service provides one server and multiple clients. The modeler configures all

of this information using our extended PNML format, which is then parsed by DiNeROS.

Listing 3 Grammer extension for System Models

```

1  DinerosTransition:Transition ::= [Ti:TransitionInfo];
2  DinerosPlace:Place ::= [Pi:PlaceInfo];
3
4  abstract PnObjectInfo;
5  PlaceInformation:PnObjectInfo;
6
7  abstract TransitionInfo:PnObjectInfo;
8  TopicInfo:TransitionInfo ::= <TopicName:String>
9  SuPort:SubscriberPort*
10 PuPort:PublisherPort*;
11 ServiceInfo:TransitionInfo ::= <ServiceName:String>
12 ServerChannel:ServiceChannel
13 ClientChannel:ServiceChannel*;
14 SignalInfo:TransitionInfo ::= /Clause:
15 InputSignalClause/;
16
17 abstract Port;
18 abstract TopicPort:Port ::= <PlaceId:String> <Limit:
19 int>;
20 SubscriberPort:TopicPort;
21 PublisherPort:TopicPort;
22 ServiceChannel:Port ::= <RequestPlaceId:String>
23 <ResponsePlaceId:String> <Id:String>;

```

5.4 Runtime model (RTM)

The RTM is a runtime model of the distributed ROS application, ensuring that it follows the SyM-modeled workflows (RQ3). An RTM is a subset of a SyM describing the behavior of one ROS node, while multiple RTMs describe the whole application and are generated automatically based on one SyM. Thus, the RTM differs only w.r.t. the service and topic elements from a SyM. All channel elements are removed except the input and output ports of senders and receivers which are added to the according places as additional information. Furthermore, tokens in ports of sending nets are automatically sent via ROS. Vice versa, ports of receiving nets are automatically filled, by the later presented runtime environment, while ROS handles the actual communication. No other transformations are needed because topics and services are the only connections between node nets.

5.5 Petri net model (PNM)

The PNM describes the ROS application using a Petri net as defined in Sect. 2, but with additional inhibitor arcs introduced within the transformations in Fig. 8. Each PNM results from multiple transformations replacing all newly introduced modeling features from a SyM by PN elements, including inhibitor arcs, enabling model checking with TINA [29]. We analyzed the ROS client library implementations of Java and C++ to enable a correct transformation of the SyM to an PNM model covering the internal communication mechanisms of ROS. Those implementations vary with regard to subscriber and server-side queuing features. Therefore, we modeled the

common subset of both, covering all RosJava features and a usable configuration of the C++ library.

In the following, we describe the transformations for each concept. They are additive, i.e., only add elements to the PN and do not remove or modify existing Petri net objects. The graphical representation of these transformations follows a rule matching pattern. The left hand side (LHS) defines the matched pattern, and the right hand side (RHS) the rule on how to replace found matches. Additionally, elements, which are underlined on LHS and RHS, refer to each other.

5.5.1 Topics

Communication requires integrating models for publisher-queuing, dispatchers, and callbacks to transform it. Those transformation rules are shown in Fig. 8. Rule $T1$ inserts for each topic element a connected place P_{topic} and transition T_{topic} representing the channel between ROS nodes. Rule $T2$ treats the input of the topic by connecting a model fragment for each publisher representing the overflowing sending queue to the publisher's output place and the channel. The queue itself has a capacity l_{pub} defined as a parameter of the SyMs topic input matched in the LHS, set as the initial marking of the capacity, which is reduced by inserting a new token into the queue via T_{pin} and increased by pushing a token to the actual communication channel by firing T_{pout} . If no capacity is remaining, then newly incoming tokens (i.e., messages) are trashed by firing the inhibitor arc. The rules $T3$ and $T4$ are transforming the outputs of topic elements by firstly inserting in $T3$ for each subscribing node N_S a receiving overflowing dispatcher queue, with a capacity l_n , which can be altered by the modeler. Within rule $T4$, for each subscribing place P_{sub} , another overflowing queue is created, whose capacity is defined within the associated topic element's input. This callback queue is attached to its corresponding dispatcher output transition $T_{subs,i}$ and to the subscribing node's place P_{sub} .

The rules $T5_a$ and $T5_b$ are the two options on how to define the actual overflow-mechanism of the inserted queues. The first option uses inhibitor arcs, which improves the resulting PNMs readability, by checking for the exhaustion of $P_{Capacity}$. The second option uses normal arcs and checks against P_{Load} . We introduced the second option as it allows the use of optimization techniques in model checking tools like TINAs *reduce*, which does not support inhibitor arcs. We will show in Sect. 8 that this leads to considerable improvements concerning the performance and memory requirements of the model checker.

5.5.2 Transforming an example

Figure 9 illustrates how a fragment of the running example is transformed into an PNM. More in detail, as shown in Fig. 9a,

the selected fragment contains two publishers $P_{SortRed}$ and $P_{SortBlue}$, originating from the selector component, publishing to the `LeftCellObjects` topic to which the controller component $P_{ReqControl_L}$ subscribes to.

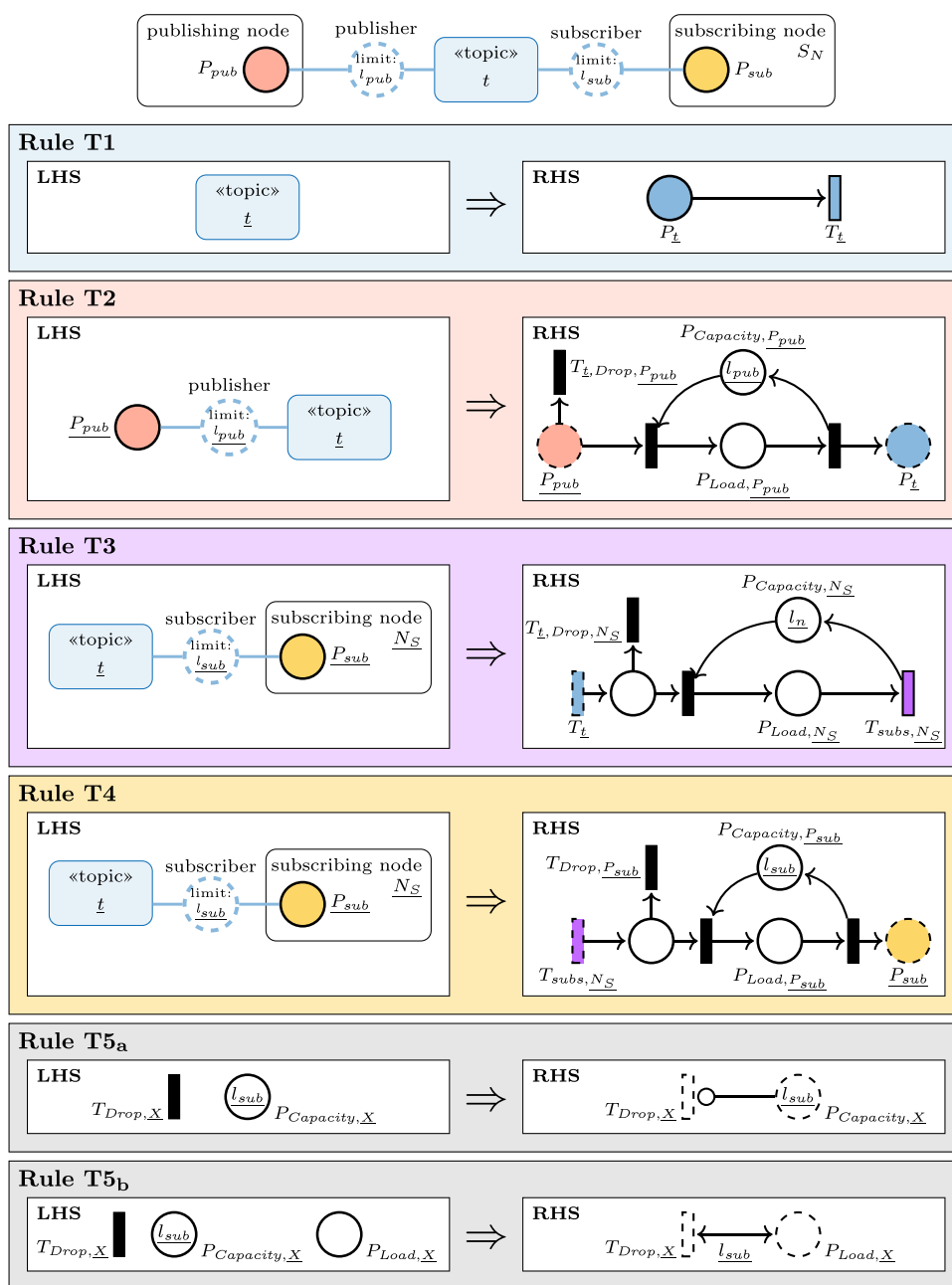
Figure 9b illustrates which rules are applied and how. More in detail, first $T1$ is utilized once for the topic channel between the nodes, followed by applying rule $T2$ for both publishers, creating the publisher queues for red and blue objects and connecting them via references to the channel. Both inserted queues provide a capacity of 10, defined by the example-defined publisher limits. Then, rule $T3$ creates the subscriber's side dispatcher queue equipped with a capacity l_n set to 16. We used 16 here, as an example as it is the default capacity in the ROS-Java implementation. Next, the dispatcher queue is referenced by applying rule $T4$, providing again a capacity of 10, defined by the example-defined subscriber limits. Finally, rule $T5_b$ is used four times to create the connections between the places P_{Load} and P_{Drop} of the various inserted queues.

The final result, shown in Fig. 9c, no longer has references and pages. This is because references are not supported by the complete TINA tooling. Therefore, the *Petri Net Flattener* removes the pages and resolves all references. Thus, the resulting net contains the directly publisher-connected publisher queues, connected to the channel, modeled by $P_{LeftCellTopic}$ and $T_{LeftCellTopic}$. This channel transmits tokens representing selections to the dispatcher and subscriber queues, delivering the tokens to the subscribing place $P_{ReqControl_L}$.

5.5.3 Services

Flattening service communication requires modeling the server-side thread pool and connections between requesting clients and their respective server instances. The corresponding transformation rules are depicted in Fig. 10. Rule $S1$ covers the communication between server instances and clients, which are attached through place references to clients. Service clients perform blocking calls, represented by a client-side place P_{ready} marked with one token removed when a request is sent and added when a corresponding response is received. The transitions $T_{req,c}$ and $T_{res,c}$ model the channel c between client and server. The SyM models a server prototype, which is replicated for each required server instance, defined by instance count C_i via rule $S4$, where the number of needed instances is computed by attributes based on a service's total number of clients. This knowledge enables fixing the connection between client and server by explicitly modeling a client's server instances. Additionally, rule $S4$ describes the interfaces of each service instance with $P_{entry,i}$ consuming requests and $P_{exit,i}$ providing the result, which is sent back to the connected client. The server side in rule $S1$ has two purple marked places, referenced in the RHS of rule

Fig. 8 Transformation rules for topic-based communication

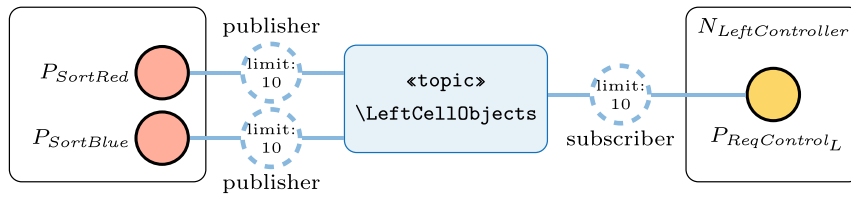


S_2 , acting as a cross-product connecting each client to all server instance’s interfaces and therefore looping over each channel c and server instances i . Each server instance communicates its status based on $P_{active,i}$ and $P_{inactive,i}$, which are referenced within S_2 and included for each server instance in S_3 . Similarly, $P_{inUse,c,i}$ provides the usage status of a client-to-server connection.

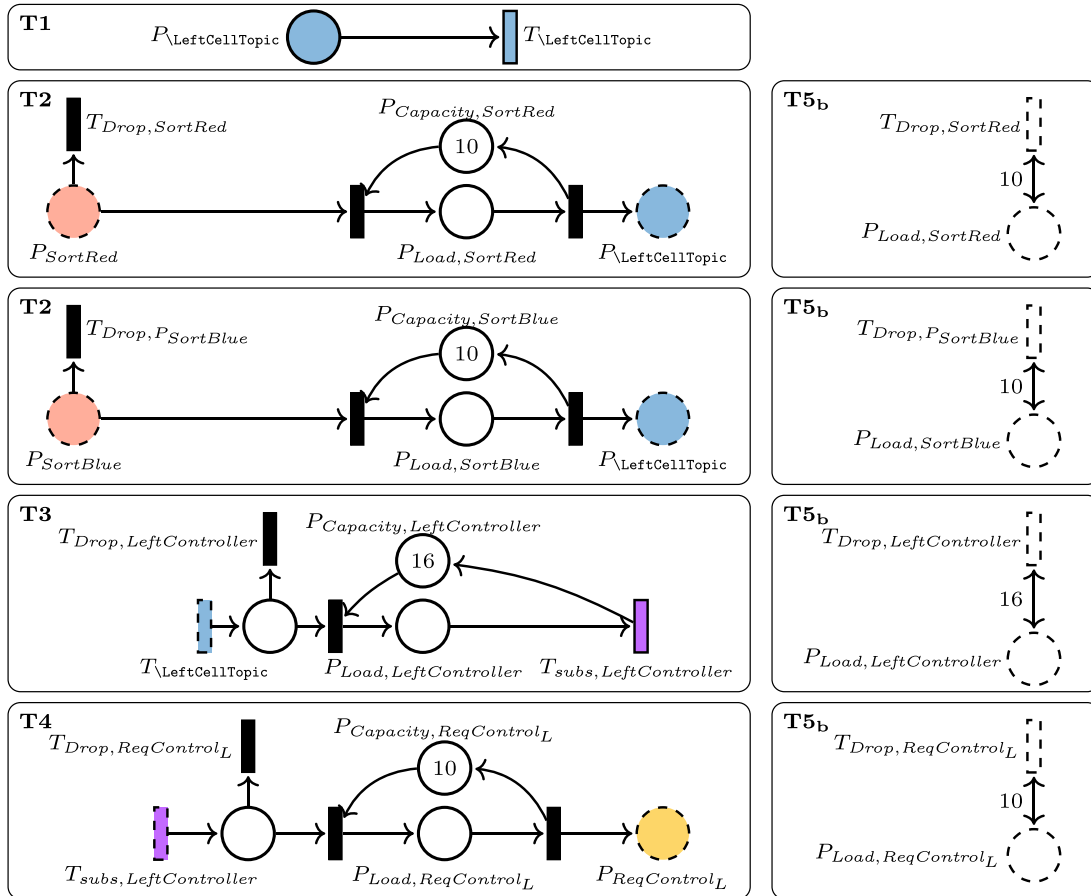
5.5.4 Signals

An *input signal clause* (IC) on a transition T_x requires its transformation to preserve its semantics within the PN,

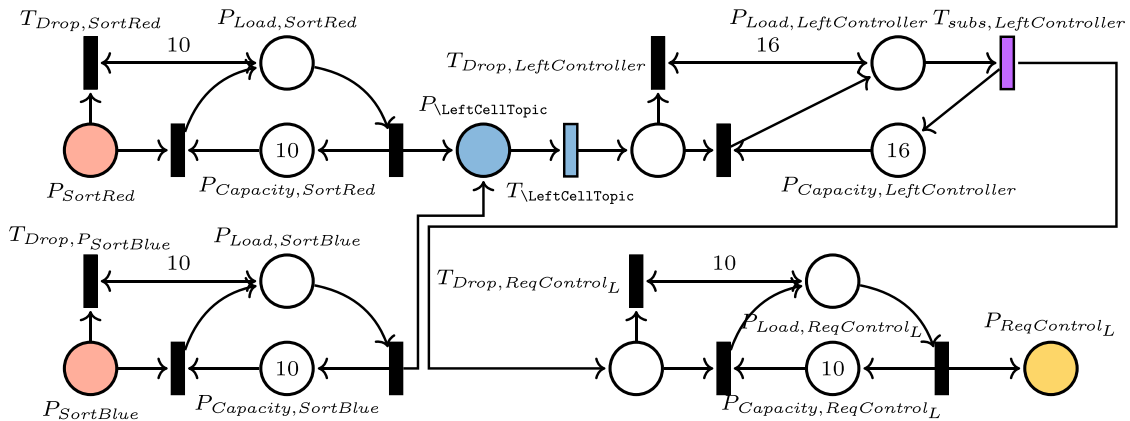
enabled by the clause in conjunctive normal form and additive transformation rules depicted in Fig. 11. Rule I_1 creates a place $P_{D_i=true}$ for each disjunction term and connects it to T_x . The signal usages, the ICs literals, are combined into these disjunctions by the rules $I_2^{fully-enabled}$ and $I_3^{fully-enabled}$ referencing the signal value changing transitions $T_{toTrue,s}$ and $T_{toFalse,s}$ on their RHS. The decision of applying either rule $I_2^{fully-enabled}$ or $I_3^{fully-enabled}$ depends on the negation of a literals value within a clause, i.e., a non-negated signal usage triggers rule $I_2^{fully-enabled}$ and a negated rule triggers I_3 . This decision is reflected in the LHS of rules $I_2^{fully-enabled}$ and $I_3^{fully-enabled}$ by matching against the literal or its negated



(a) Matching parts of Fig. 14



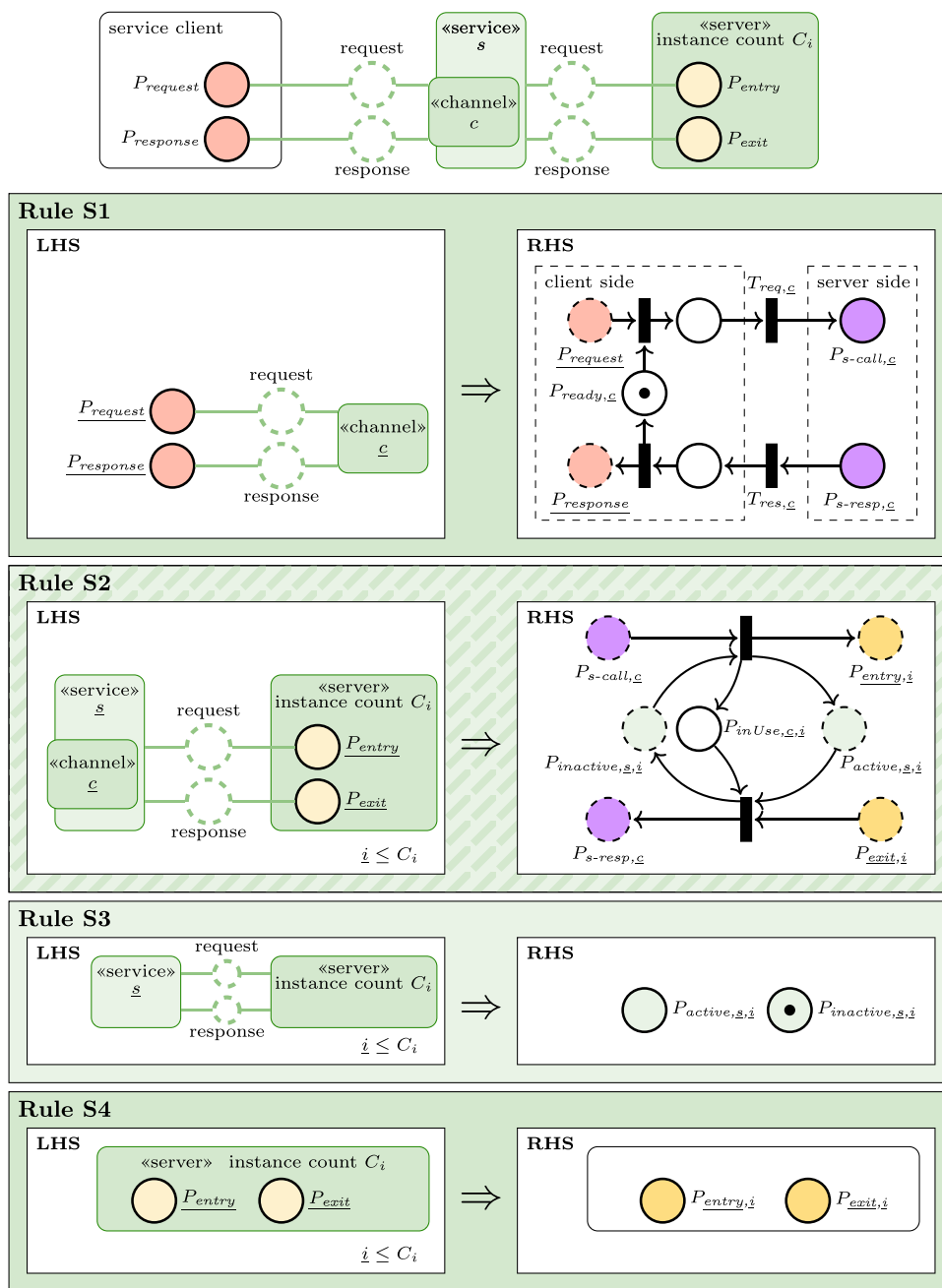
(b) Pages resulting from rule application



(c) Flattened version of Fig. 9b (PNM)

Fig. 9 Flattening of a topic with two publishers and a subscriber

Fig. 10 Transformation rules for service-based communication

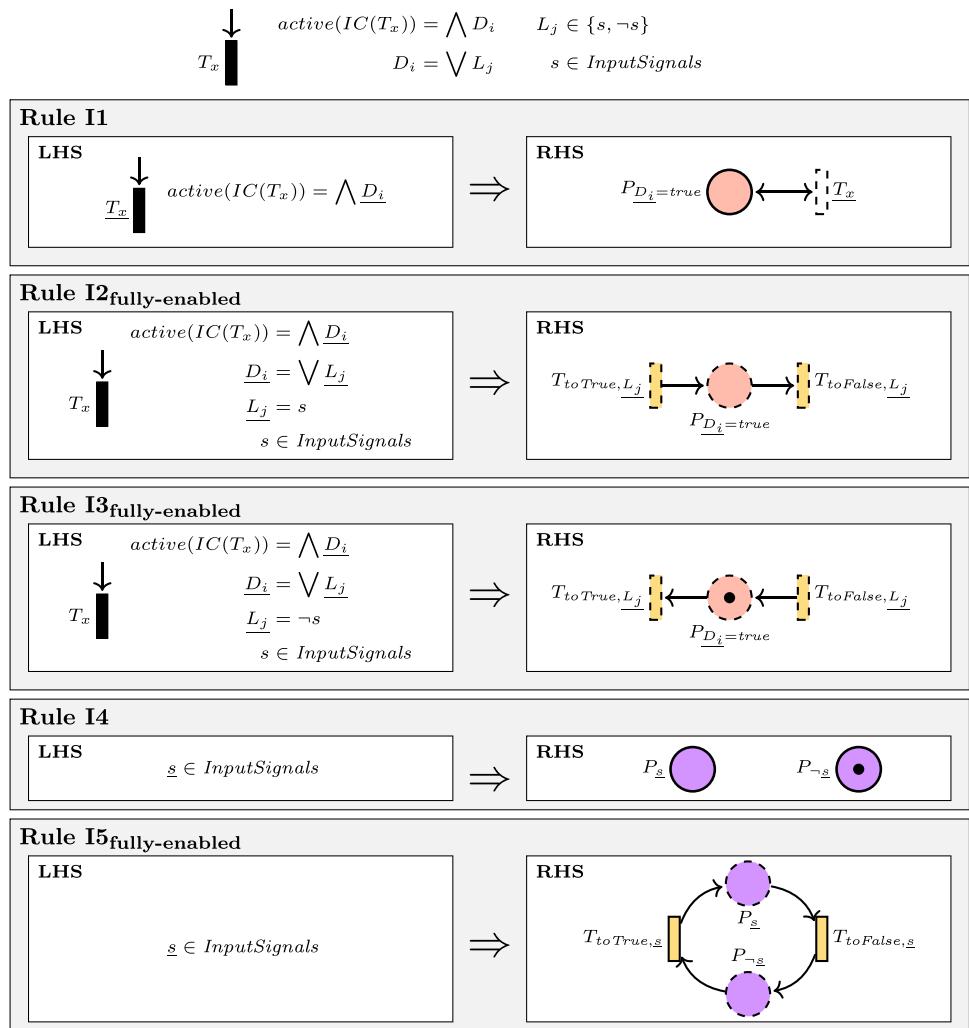


version. Rule *I4* is applied for each input signal s and inserts places for both states of an IS: P_S for *true* and $P_{\neg S}$ for *false*, representing literals of a clause. The rule *I5_{fully-enabled}* connects these places, allowing a signal value change. The rule *I5_{input-enabled}* connects these places, allowing a signal value change.

The results of the previously described transformations are referred to in the following as *fully enabled* signals, because they can change their value at any time, justified by the fact that $T_{toTrue,s}$ and $T_{toFalse,s}$ just depend on the signals current state. This increases the state space by two to the power of signal usages. Thus, we introduce two optimizations here.

The *input-enabled* variant is described in Fig. 12, where rule *I5_{input-enabled}*, creates for each signal usage new transitions T_{toTrue,s,T_x} and $T_{toFalse,s,T_x}$, and links them to the signal state defining places. The new rule *I6_{input-enabled}* bidirectionally connects the input places of T_x to the aforementioned transitions, whereby the corresponding signal model fragment marking now only changes when the inputs of at least one of this signal using transitions are enabled, reducing the resulting state space. The original rules *I2* and *I3* are also modified for the *input-enabled* variant to reflect the per-signal usage included in rule *I5_{input-enabled}*.

Fig. 11 Transformation rules for input signal clauses using fully-enabled signaling



The *io-enabled* variant, extends the *input-enabled* variant by additionally creating transitions $T_{toTrueOut,s,T_x}$ and $T_{toFalseOut,s,T_x}$. These transitions are additionally bidirectionally linked to the output places of T_x . Although this variant allows more changes to the signal value, it describes the actual behavior of a signal more closely, as it can reset its value after it has been introduced into the system.

5.5.5 Distribution

In DiNeROS, communicating node nets are independently executed. The firing of transitions in different node nets is performed in parallel, potentially creating race conditions. However, no such conflicts can happen since node nets are decoupled using only topics and services.

6 Modeling the running example

To evaluate that our proposed approach is feasible and enables the detection of bugs, we first describe the SyM

of our case study, followed by scenarios in which we used path analyzers to detect erroneous behavior introduced by unaligned changes of different developers. Finally, we analyze the use of the resulting runtime models, which are obtained by removing the SyM-introduced channel elements as described in Sect. 5.4.

The physical representation of the case study involves two industrial robot arms, shown in Fig. 13. Both arms collaborate within a fixed zone to sort objects, while a human operator can interrupt the robots. The corresponding SyM is shown in Fig. 14. It represents node nets modeling the components described in Sect. 4. We use balloon tokens to capture information about the color, pose, and name of objects and for status information, namely if picking and placing succeeded. The flag `humanDetected`, used in Listing 2, stores sensor information. If a token goes through the sensor component, it is enriched with sensor information. The selector component provides places 1 for each color with tokens for the objects. These tokens are consumed when the attached transition fires, which are controlled by individual ICs representing the user

Fig. 12 Transformation rules for input signal clauses using input-enabled signaling

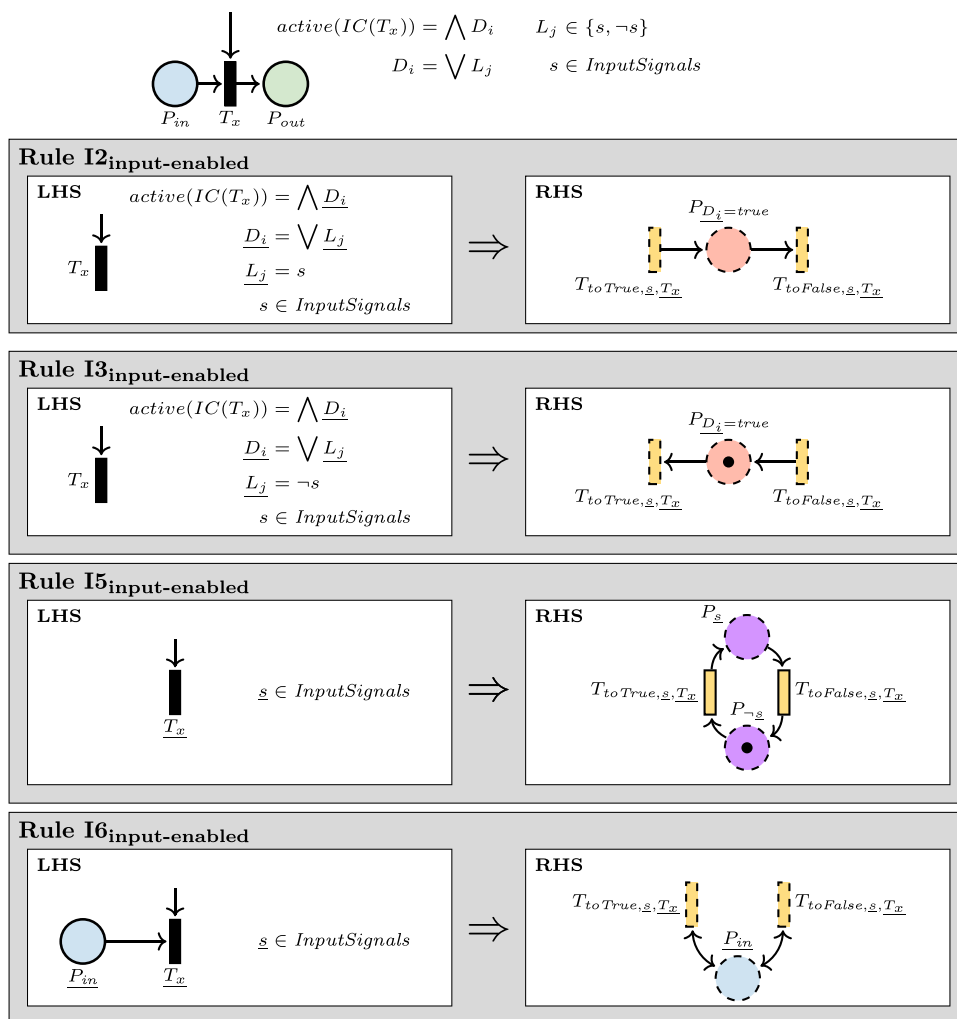


Fig. 13 Physical setup of our case study



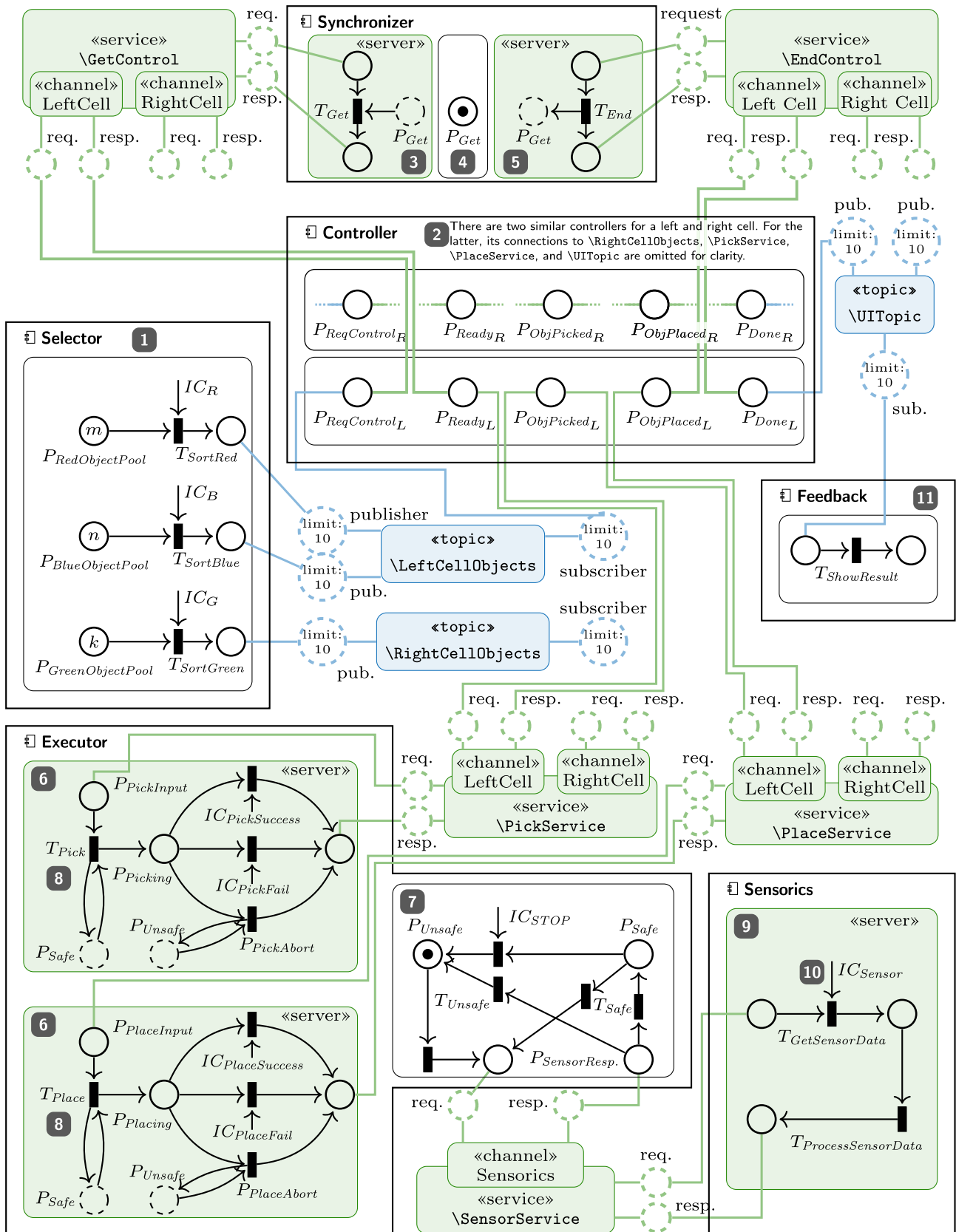


Fig. 14 Running example as SyM, where e.g. $IC_{PickSuccess} := (RobotIsIdle \vee RobotHasFinished) \wedge ObjectIsPicked$

input. Depending on the color, the selector net sends objects via topics to the controller, where two identical robot controllers **2** are sorting the objects based on the topic they listen on. These controllers represent the central workflow describing the steps to sort an object and coordinate the other components. A robot-controller's workflow starts with requesting access to the robots' shared workspace, picking objects, placing them into bins, leaving the shared workspace, and releasing the control over it after the $P_{ObjPlaced}$ place indicates that this pick-place process has finished. Requesting and releasing control is done based on two services **3/5** within the node net of the synchronizer component. The exclusive access to the workspace is represented by the place P_{Get} within the control state page **4**, connected to the instances of the control requesting and releasing services based on references. This resource can only be acquired by a single `GetControl` server instance at a time because there is only a single token within $P_{StateGet}$.

The actual picking and placing actions are realized by service calls in the controller nets, where the corresponding servers represent the executor component within a ROS node. This component consists of the application logic **6** and a shared safety model **7**. Sharing the safety model is realized by letting the pick and place services access it with place references. The first transitions T_{Pick} and T_{Place} of the executor's services access the safety model based on references and lets a robot pick respectively place an object by its attached callback when it is safe **8**. The result-tokens of pick and place operations are output in a respective place. These tokens are processed further depending on whether signals indicate failure or success, or the state of the safety model.

The safety model queries the sensor component **9** to check if the current environment is safe, which can either result in a token in P_{Safe} or P_{Unsafe} , aborting execution within the referencing pick and place services. The sensor component is represented by a node net retrieving data if a detecting sensor is active. The availability of new data itself is indicated by the attached signal clause IC_{Sensor} **10**, solely defined by a single signal named *Sensor*. Finally, the controller node nets sends the result of their workflow execution via the topic `UITopic` to the node net controlling the user interface **11**. For our evaluation, the previously defined application was developed with DiNeROS by a team of developers splitting responsibility for the different application components.

7 Analyzing DiNeROS applications

Various analyzes can be carried out on the basis of a PNM and its statespace. TINA [29] provides basic algorithms, such as (partial) state space generation and path search, which aid in analyzing PNMs. The *Net Analyzer* uses these algorithms

for domain-specific analysis methods in DiNeROS. These analysis methods are presented below.

7.1 Analyzing state spaces

Whenever a message, represented as a token, is lost within a PN-based ROS application, this affects all dependent Petri net parts. Detecting erroneous communication constructs for topics is possible by a path analysis when looking at the overflow transitions of publishers, dispatchers, and subscribers of a PNM. Every time one of these transitions (T_{Drop} in Fig. 8, rules T2–T4) is enabled, a queue will overflow. In Algorithm 1, we generate the state space of a PNM until a state satisfies this overflow condition. This is done for all topic channels (Line 2) and their containing overflow transitions (Line 3) on the publisher and subscriber side. Thus, the algorithm covers all potentially overflowing transitions.

Algorithm 1 Analyzing topic channels

```

1: procedure ANALYZEOVERFLOWS(model)
2:   for channel in TOPICCHANNELS(model) do
3:     for  $t_{overflow}$  in FINDOVERFLOWTRANSITIONS(channel) do
4:       violating_state ← sift -R -k -f "- ({$t_{overflow}});" -df $model
5:       partial_space ← sift -R -k -c $violating_state -df $model
6:       trace ← pathto -p $violating_state $partial_space
7:       VISUALIZETRACE(trace)
8:     end for
9:   end for
10: end procedure

```

Within the first step on Line 4, we use *sift*, which is part of TINA and enables the construction of reachability graphs (i.e., state space) and on-the-fly verification of reachability properties. More in detail, the `-f` option is used to specify in the following a stopping condition for the state space generation, where we stop if an overflow transition is enabled. The `-df` option uses a depth-first approach in our example, but depending on a Petri nets structure, a breadth-first approach could result in a faster termination. Additionally, the TINA tool *selt* can be used to check any linear temporal logic (LTL) formula. The second step on Line 5 creates a partial state space contain the previously identified invalid application state, by using the `-c` option. This partial space is used by TINA's *sift* tool, creating a trace from the initial marking to the violating state. The trace is analyzed and visualized by the *Trace Visualizer*, simplifying development using Petri nets as formalism (RQ1). By this, we avoid generating the complete state space, which gets impractical for growing Petri net sizes.

Within Algorithm 2, the state space of a PNM is examined to find and analyze unused parts of it, allowing it to either repair or minimize the net afterward. The algorithm assumes the satisfiability of all signal clauses defined within the net. Therefore, all model fragments defining the signals and their bindings are removed on Lines 5 to 7. Additionally,

Algorithm 2 Analyzing dead transitions

```

1: procedure ANALYZEDEADTRANSITIONS(model)
2:
3:   problems ← []
4:
5:   for signal_fragment in GETSIGNALFRAGMENTS(model) do
6:     REMOVESIGNAL(signal_fragment)
7:   end for
8:
9:   deadTransitions ← GETDEADTRANSITIONS(tina -s 0 $model)
10:
11:  for t in deadTransitions do
12:    if ISSERVICECALLTRANSITION(t) then
13:      problems ← (GETSERVICE(t) is never called)
14:    else if ISSERVICERESPTRANSITION(t) then
15:      problems ← (GETSERVICE(t) never returns)
16:    else if ISTOPICCHANNELTRANSITION(t) then
17:      problems ← (GETTOPICCHANNEL(t) is never used)
18:    end if
19:  end for
20:  return problems
21: end procedure

```

it is assumed that, based on the results of Algorithm 1, overflows on topic channels have been detected and resolved. On Line 9, the output of TINA, which is the state space of an PNM, is used to get the list of all dead transitions. This list is iterated over (Lines 11–19), and based on the contained transitions and types, it is determined whether a topic is used, a service is never returned, or a service is getting called.

Algorithm 3 also examines the TINA-generated state space (Line 2), but checks whether a node defined by a given model_fragment is ever used (autonomously or by call). This involves checking whether all transitions in the model_fragment, i.e., including communication interfaces, are dead (Line 4–10). If a node is dead, it will either never be called, or its transitions will not fire on their own. This is an indication of either a modeling error or an obsolete node.

Algorithm 3 Checking of node usage

```

1: procedure ISNODEDEAD(node_fragment)
2:   deadTransitions ← GETDEADTRANSITIONS(tina -s 0 $model)
3:   matchCount ← 0
4:   for deadTransition in deadTransitions do
5:     for transition in TRANSITIONSINNODE(node_fragment) do
6:       if transition equals deadTransition then
7:         matchCount ← matchCount + 1
8:       end if
9:     end for
10:  end for
11:  return matchCount == NUMTRANSITIONS(node_fragment)
12: end procedure

```

7.2 Signal-caused effects

The satisfiability of signal clauses has a major effect on the accessibility of net parts. When examining Fig. 14, a

first example can be created by defining the clause IC_{STOP} in 7 as $Danger \wedge \neg Danger$. This clearly disables the bound transition permanently, where $Danger$ is initially *false* and updated at runtime. This would make the application unsafe because, when having the response token of the sensor components service in P_{Safe} , it is impossible to reach P_{Unsafe} again. Additionally, the *SensorService* is thereby also called a maximum of once. This defect was detected by analyzing the sift-generated reachability graph of 8 and 10, showing that the IC_{STOP} transition is never, and transitions in 10 are only enabled once. Detecting such erroneous constructs is done by iterating over all signal clauses and investigating their satisfiability, by using a SAT-solver, and reporting not satisfiable clauses as list.

A second example is defining the clause $IC_{PickSuccess}$ as constantly evaluating to false, preventing the pick service from returning, essentially blocking all robots. Such properties can be expressed as LTL formulas and checked with *selt*, where the only content of the formula is the identifier of the services P_{exit} (see Fig. 10). Hence, this formula checks if the transition is ever enabled, which is not the case if the service ever responds. Thus, the influence of signal-based attached components and models is analyzable (RQ2).

8 Scalability of analyzing DiNeROS nets

One of the advantages of Petri nets is that they can specify a large state space in comparison to the model size. However, this also implies that the generation of state spaces consumes more time and memory as the model size increases. The state space of a PNM grows even faster. This is firstly because of the usage of signals. When using the fully enabled version of signals, we increase the number of states by 2^{su} where su is the number of signal usages on transitions. Secondly, the usage of queues within topic channels increases the state space. Therefore, we performed an evaluation of the scalability of the state space generations in relation to the signal flattening variations. Within the evaluation, the results of the three types of signal transformations were compared with a model not using signals (*without signals*). We measured each of these types both with and without reduction by TINAs *reduce* tooling. Especially, as shown in Algorithm 4, we use a two-stage approach. First, in Line 2, a reduction is performed that preserves the reachability set and removes duplicate places and transitions as well as identity transitions [46]. Within the second stage on Line 3, we use the `tr` algorithm to compute clusters, reducing the net even more. This algorithm always reduces the input nets of our running example in less than one second. We use a desktop PC with a Ryzen 9 3900X and 64 gigabytes of RAM running Ubuntu 20.04 as the measurement system. The measurement of time and memory consumption values was conducted with the

Unix *Time* tool. More specifically, the Non-Bash-integrated version of this tool was taken, as this allows more accurate measurements of memory usage. Unfortunately, the accuracy of time measurements is limited to 10^{-2} seconds. The reason for using an external tool for measuring is that the TINA source code is not publicly available.

Algorithm 4 Reduction of a PNM with TINA

```

1: procedure REDUCE(pn_model)
2:   rg_reduced ← reduce -rg,redundant,compact $pn_model
3:   tr_reduced ← reduce -tr,cluster,4ti2 $rg_reduced
4:   return tr_reduced
5: end procedure

```

Figure 15 shows the number of states and transitions of the generated state space depending on the number of tokens used in the object pool places from our running example (i.e., the number of colored objects subject to sorting). The diagram shows two variants of four alternative approaches. The default case (fully-enabled) was introduced in our previous work [24]. Next, we introduced two optimizations to reduce the state space of signals by modifying the result of the signal translation shown in Fig. 11. The first, called io-enabled, restricts the transitions of signals to be only enabled when there is a token in either an input or an output place of the original transition. That is, each transition T_{ioTrue,L_j} and $T_{ioFalse,L_j}$ gets each input and output place of T_x as input place. This reduces the state space of the overall net. The second optimization is called input-enabled and restricts the signal transitions further by just enabling them when there is a token in an input place of T_x . The rationale behind these two optimizations is that a signal typically does not change when the respective part of the net is not currently active. Additionally, we added *without signals* as a reference version of the case study without signals. For each of these four approaches, we show the variant with and without TINA reduction as shown in Algorithm 4. Some measurement sequences do not capture the whole range of objects to be sorted, because in these cases the measurements could not be completed as the measuring systems RAM was exceeded.

The first insight that can be observed is that – compared to the fully enabled and not reduced version proposed in the preceding work – the io-enabled version reduces the state space by one magnitude and the input-enabled by an additional one. This is possible, because with these modifications signal transitions are not always enabled, but only if there are tokens in the inputs and/or outputs of the connected transitions of the model. Reducing the io- and input-enabled version based on Algorithm 4 reduces the size of the state space successfully. Both are not as performant as the fully-enabled reduced variant, because the reduction step for the fully-enabled variant completely removes the signals. The additional arcs to enable

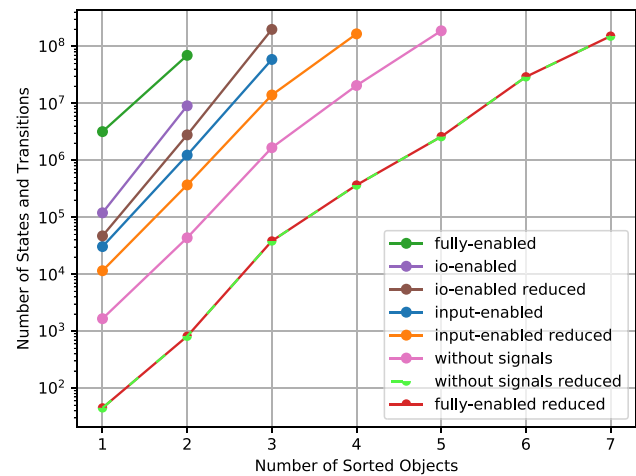


Fig. 15 State space size (states) dependent on used approach

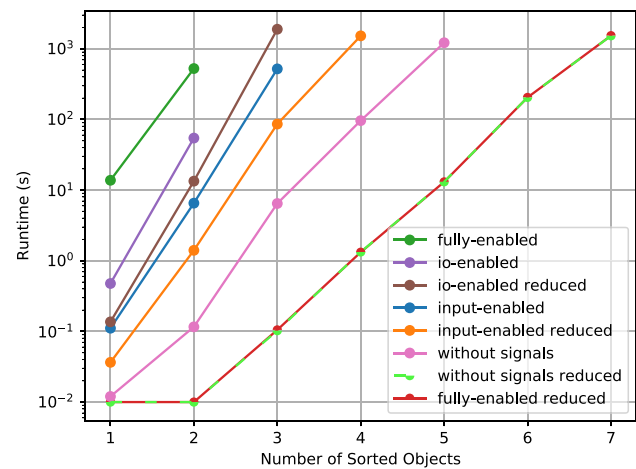


Fig. 16 Time consumption of the state space generation

signal switches for in- and output places prohibits this optimization.

The second interesting insight is about comparing the fully enabled approaches result with its reduced version and the model that does not use signals. It can be observed that both, reducing the fully enabled approach, and the version without signals generate spaces of the same size. This is because the reduction removes signals in the fully enabled model, as they can always fire there and only introduce redundancy in terms of the state space. Therefore, after the removal of signals the non-signal and fully enabled version can be reduced in the same way, resulting both in a four magnitudes less large number of states and transitions.

In addition to examining the number of transitions and places in the state space, the memory consumption and the time required to generate the state space were also considered, within Figs. 17 and 16. When measuring the times, they are limited to 10^{-2} , because they are below the measurement accuracy of the used tooling.

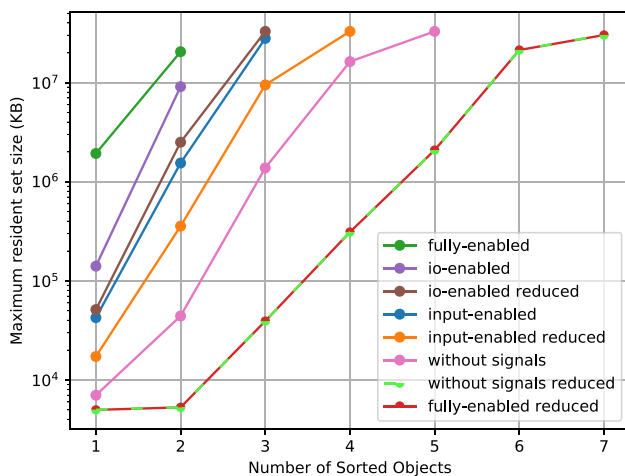


Fig. 17 Memory usage of the state space generation

Both diagrams show that we can observe a trend in memory and time consumption, where in both cases, the additional memory or time required increases as the number of objects to be sorted increases, especially when considering signal-less and reduced models. Thus, if a transformation reduces the state space by several magnitudes, more tokens can be included in the state space generation. Nevertheless, the increase in time and memory consumption is exponential. Reduced fully-enabled and signal-less models have nearly the same memory consumption clearly justified therein that their state space shares the same number of places and transitions, and the models themselves follow a nearly identical structure. In terms of memory consumption, there is a noticeable aspect related to the fully enabled and signal-less reduced variants, both sharing the lowest memory usage. Both variants have a strong increase in memory usage from two to three objects to be sorted, compared to the directly following measurements. This is because the reduction has a stronger effect on the first two scalings, as the first two tokens only flow into the `LeftCellObjects` topic, and therefore, a significant part of the selector component can be reduced away, which is not the case for the third scaling anymore. This effect is still visible even in the unreduced variant without signals, as the signal-less state space grows more slowly, due to the non-use of a topic. By not reducing and using input- or io-signals, this effect is not present in the other measurement series. A second noticeable aspect, shared by the four right-hand memory measurement series, is about their respective two last scalings, in which the increase in memory growth decreases. This is because less memory is allocated by TINA when the RAM is close to exhaustion, while more optimistic allocations can be made in the other scalings.

Generalization of findings

The results of analyzing the scalability of the individual measurement series are specific to the running example. However, some findings can be generalized. First, the input-enabled and io-enabled versions always reduce the state space, regardless of the model used. This is important, because the improvements of the *reduce* tool might only work that well for the example shown in this paper. Other Petri nets might not benefit that heavily from using the *reduce* tool, but will benefit from using the input- or io-enabled versions. In our example, the signals are recognized by the optimizer as irrelevant to the state space in the reduction, while in general, the signals could be considered relevant. Second, the location of signal usages is important. When signals are used in front of communication channels or more complex sub-networks, the state space increases considerably more compared to the case where signals are used in less complex sub-networks. Finally, the distribution of the Petri net to node nets and their internal net structure influences the efficiency of the *reduce* tool.

9 Debugging the running example

The detection of erroneous models can take place in DiNeROS at modeling time on the one hand and at application runtime on the other. This is illustrated below using the running example, where we use the io-based flattening of signals, reducing the potential number of signal value changes and thus improving the readability of traces to be analyzed.

9.1 Analysis of traces

An important tool for detecting errors in a DiNeROS application is the analysis of traces as introduced in Sect. 5.2. These traces are always only one possible run-through and, therefore, only represent a part of the overall picture. Within DiNeROS, we are able to analyze two kinds of traces: TINA-generated modeling-time traces and runtime execution traces generated by the Petri Net Engine. The following section takes a closer look at the analysis of modeling time traces.

A modeling-time trace is generated as shown in Algorithm 5 by searching a goal state where a given transition is enabled with *sift*, generating the partial state space and building a trace to the goal state with *pathto*. In the use case, if we want to know when an object has passed through the workflow, the goal state corresponds to the last transition *T>ShowResult*. The complete PNM of an application is used for this, as signals are necessary to obtain a complete understanding. As already determined, the state space grows strongly. Therefore, partial state spaces are used for modeling-time traces.

Algorithm 5 Creating a trace for a sorting operation.

```

1: procedure CREATETRACE(model, goal_transition)
2:   goal_state ← sift -R -k -f "- (($goal_transition));" -df $model
3:   partial_space ← sift -R -k -c $goal_state -df $model
4:   return pathto -p $violating_state $partial_space
5: end procedure

```

A modeling-time trace of our running example is shown in Fig. 18, where a first (red) object was sorted and the process for a second (green) initiated. First, on the x-axis, we can observe the number of fired transitions, while the y-axis is used to indicate information about the state of signals. Second, below the x-axis, additional information about topic usages and selected transitions with attached handlers are displayed. Third, the use of services is indicated by curly brackets, which provide information about a services name and the identifier of communicating service instance (*I*) and client (*C*). The brackets start at the service call and end when the service returns.

This visualized trace of the running example starts on the left-hand side with the first execution of the sensor service, called by the safety model. This call is repeated over the entire trace, because the trace is built using depth-first search. Moreover, in principle the sensor is always allowed to be called, when having a token within the safety models calling place, because only the pick-and-place workflow is dependent on the safety model, not vice versa. More in detail, the sensor signal marked in yellow changes its state several times, e.g., when the sensor is called up for the first time. This is possible because with the io-enabled approach a signal's state is allowed to change before and after firing the attached transition, i.e., when there are tokens in the transition's in- or output places. TINA exploits all possible traces, i.e., all allowed sequences of signal states, service calls, and so on. The visualization shows one of many possible traces. Next, the sorting signal for *Red* objects is activated, followed by the signal for *Green* objects. The *Blue* signal is never activated because no blue object is included in the example. Selections of objects are sent using the `LeftCellTopic` and `RightCellTopic` to the controllers, where a controller acquires control via the according service and calls the `Pick` service. After the `Pick` transition fired, the `PickSuccess` signal indicates the operation's success. Two callbacks are called. One for the `Pick` transition and one for `PickSuccess`. Both callbacks are called relatively late in the pick-services call, as transitions of safety model and sensor service are fired beforehand.

This is followed equally for the place operation and by releasing control over the shared space via the `EndControl` service call, which allows the second `GetControl` call to proceed. The trace ends with reporting the result of sorting the red object to the UI-component via the `UITopic`.

9.2 Analysis of the state space

In the following, we look at analyzes of the state space using the analysis algorithms from Sect. 7.1, for two scenarios. Furthermore, the model used permanently enabled signals to reduce the size growth of the state space.

9.2.1 Scenario 1: Sorting more types of objects

We start with one type of object to be sorted and have a detailed look at the topic-based communication between the selector and one of the robot controllers of the initial model. Because the Petri net is 10-bounded, we do not generate an overflow when assigning capacity 10 to input and output of the selector topic and a maximum of 10 tokens to the source place of the selector. Assume a new developer extends the selector component with two more types of objects, setting the number of objects for each type and the topic's input capacities to five. However, the robot controller's developer does not modify the output capacity, which is high enough individually, but not in total. This scenario leads to a state where the corresponding overflow transition is enabled. We can detect these potential overflows by using Algorithm 1, without the *Trace Visualizer*, because the visualization does not contain useful information in this context.

9.2.2 Scenario 2: Changing signal-clauses

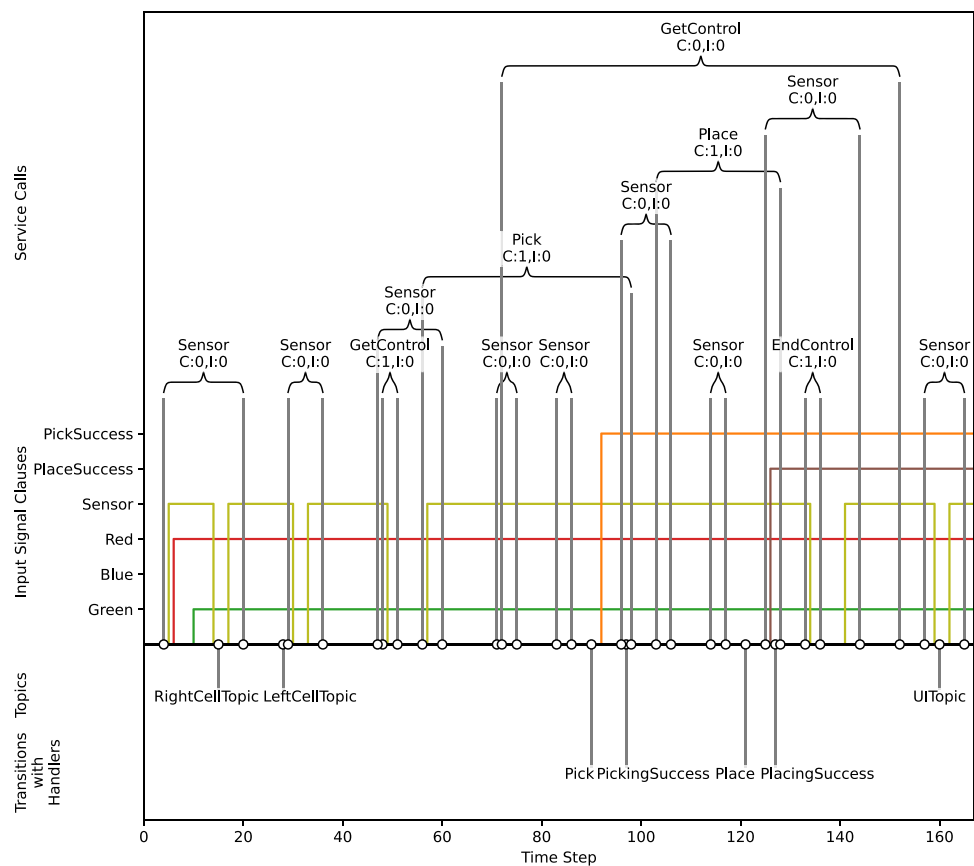
The second scenario deals with the effect of changing the clause IC_{Sensor} within the sensorics component to an unsatisfiable variant by the developer of this component without using an SAT solver to detect the unsatisfiability. This causes the sensor service to never return again and, thus, to never reinsert a token into the safety model, which again leads to the effect that the pick operation never returns to the controller component, ultimately blocking the complete application. We can detect this by using Algorithm 2. The algorithm's result lists the `Sensor` service and `Pick` service as never returning, the `Place` service as never called, and furthermore, the `UITopic` channel as unused.

9.3 Examining the runtime behavior

Beside using Petri nets within our approach at modeling time for verification purposes, Petri nets are also used as models at runtime, enabling system monitoring. In the following, we provide a detailed answer to research question 3.

First, the non-determinism of Petri nets must be handled, which is done using a handler, as shown in Listing 2, and defaults to an eager firing logic, firing a random enabled transition whenever possible. This eagerness is essential for safety in robotic applications, e.g., in Fig. 14 7, where the execution logic ensures the safety mode is adapted as soon

Fig. 18 Trace of picking and placing an object, based on a generated state space



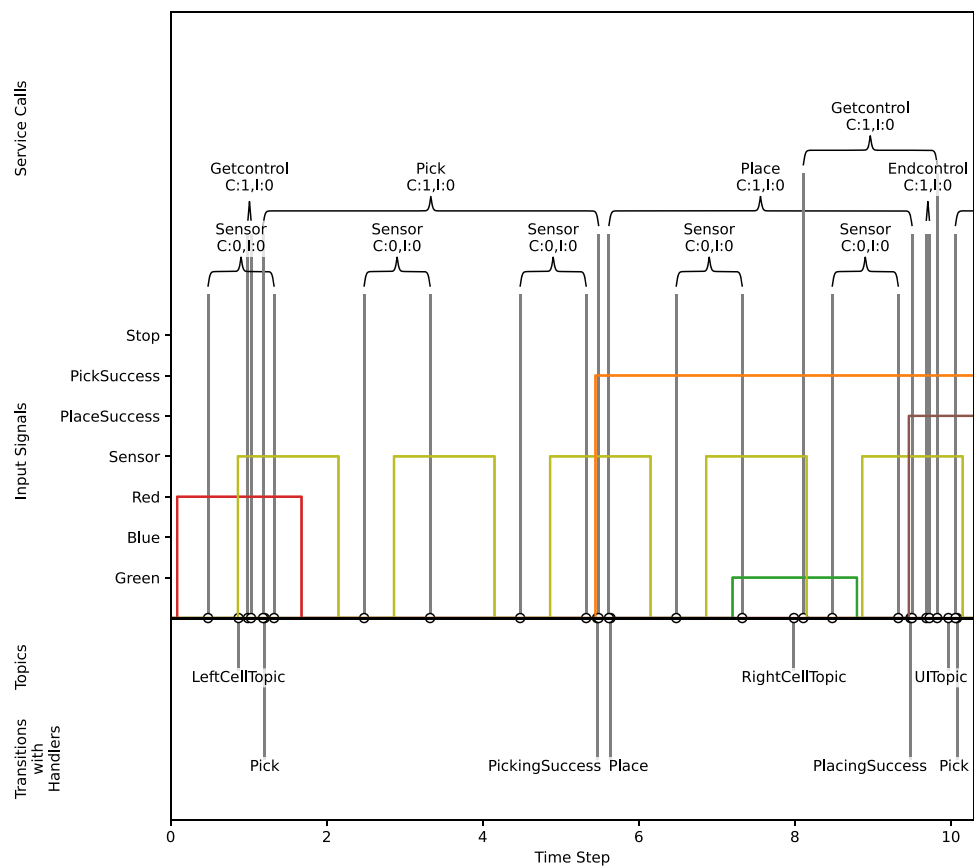
as possible, and the non-determinism is resolved as shown in Listing 2. Second, the data flow modeled with balloon tokens is specified in the transition handlers (as shown in Listing 1), computing the output token balloons based on the input tokens. Third, the causal connection to other ROS components, a defining feature for models at runtime [23], is established using callback handlers for outgoing and signals for incoming communication.

Fourth, at runtime, the execution of the RTMs of each node is handled by a node controller responsible for firing transitions and calling handlers. In conclusion, the runtime behavior is derived from the Petri net, with non-determinism, data flow, and connection with other systems defined by the user.

Finally, a runtime execution trace, the second type of trace, is generated by saving the transitions fired by the application as well as the originating and resulting states within a trace file, which is continuously evaluated at runtime by the *Trace Visualizer*. Runtime execution traces are displayed in a similar way as modeling-traces, which is shown in Fig. 19. However, the x-axis now displays information about the actual time spent by the application. Additionally, information about the usage of service instances is already partially

shown when a token is still active within an instance and updated at each new generation of the visualization. The exemplary trace in Fig. 19 shows again the sorting of a red and a green object. Similar to the modeling-time trace the sensor service is called at a certain interval. However, the sensor signal does not flicker as much because we have implicit priorities due to the handler and callback connected code. Another difference is that a user chooses the time of object color selection and does so with a greater time interval. Consequently, it can be seen how after the value change of the red signal, and the sending of the selection to the controller via the *LeftCellTopic*, control is acquired and the picking starts, resulting in the *PickingSuccess* signals value change and the returning of the *Pick* service. During the place operation of the red object, the sorting of green objects is finally started by the user, which is submitted to the right cell's controller via the according topic. The place services node net, receives the new value of the *PlacingSuccess* signal after the robot has successfully finished placing and after executing the bound transition, returns to the controller. Just like the modeling-time trace, the runtime trace ends with the first picking controller ending the control over the shared space and reporting the results using the *UITopic*.

Fig. 19 Trace of picking and placing an object, based on an application execution



10 Discussion

In the following, we discuss the practical and theoretical implications of the proposed approach, as well as considerations regarding its feasibility and generalizability.

There is always the possibility of introducing faults through hand-inserted code on the levels of handlers and callbacks despite correct model verification. Thus, a solid testing method is crucial for any additions of handwritten code. Nevertheless, the overall control is always correct with respect to its Petri net-based specification. The correctness of the model-based specification relies on the expertise and knowledge of the modeler. Hence, despite the model checking methods in DiNeROS there exists a risk of incompleteness or ambiguity in the specification.

The growing model complexity and size is a challenge in our research framework. As the model evolves and expands in scope, driven by the requirements of the system subject to modeling. That's why we use pages as means to separate concerns such as nodes, groups of nodes and net parts of nodes, enabling switching between abstractions. Nevertheless, the complexity of the system model specification itself is a barrier, especially without graphical modeling. Hence, we are currently developing a visual editor to circumvent this barrier.

The learning curve associated with our modeling framework is steep. The concepts of Petri nets and the Robot Operating System (ROS) required by modelers and developers can be learned relatively quickly. The challenging part of the learning curve lies in analyzing the Petri net model. This analysis requires detailed knowledge of Petri net-based model checking. For this reason, we introduced the separate role of the system engineer. Additionally the proposed tools within DiNeROS, such as *Petri Net Splitter* or *Package Generator* are automated, allowing users to focus on modeling, implementation and model checking.

Looking at the formal analysis methods and the growth of the state space, the resulting limitations become apparent. Successful generation of the complete state space of a model becomes increasingly unlikely with increasing model size and complexity. This is the reason why we only generate partial state spaces in this work, even if this limits the amount of guaranteed properties, such as invariants.

Regarding the engine, no guarantees can be made regarding real-time properties because the Java client library for ROS does not support them. Therefore, these are not implemented in the engine. However, ROS allows DiNeROS to be used for any robotic platform, as robotic components can be attached based on handlers and signals.

The case study reflects essential aspects contained in ROS applications: chained topic and service calls, distributed coordination and access to hardware (robots, sensor). However, the case study's scale may not fully reflect large-scale ROS applications because it operates within a simplified scenario that focuses on the investigated aspects. Furthermore, the controlled nature of the case studies environment may lack the unpredictability of real-world ROS applications. In other words, our small-scaled case study is beneficial for internal validity as we eliminate external influencing factors, but it diminishes our external validity in terms of generalizability.

11 Conclusion and future work

In this paper, we introduced a novel model-driven toolchain to develop robotic applications based on the Robot Operating System. In our approach, first, modelers design the application using an extended Petri net language, the System Model (SyM), which offers dedicated language constructs for ROS-based applications. This answers RQ1, i.e., how to develop ROS applications, which can be formally verified in terms of communication defects, influence of external applications, unused parts and the node internal workflows based on TINA. Input signals can be specified within the SyM to enable integration with other formalisms, which answers RQ2. Second, system engineers use these models to verify the formal properties of the designed applications. To enable using off-the-shelf verification tools, the extended Petri nets are transformed into basic Petri nets. Finally, the SyM is split into Runtime Models (RTMs) for each ROS node, of which each is used to generate ROS code with dedicated extension points to be used by developers. The final application uses these RTMs at runtime and invokes the code provided by developers, which answers RQ3. We showed the feasibility of our approach using an industrial case study where two robot arms collaboratively sort objects in a shared work environment. To show the usefulness of Petri net verification, we presented three scenarios. In summary, our approach improves the development of safe ROS applications due to the integration of verification activities and MDD with code generation combined with runtime models.

The presented work offers the opportunity for numerous extensions in connection to the ROS landscape. Firstly, for scope reasons, ROS actions were not addressed in this work, as these are based on topics and are, therefore, implicitly supported by DiNeROS. Furthermore, the states of an action server can be modeled with Petri nets. Secondly, a central extension of this work will be the modeling of the ROS successor system ROS 2, which provides quality of service (QoS) properties in the area of communication. For this reason, the use of Time Petri Nets will be necessary.

Acknowledgements Funded by the German Research Foundation (DFG) as part of Germany's Excellence Strategy - EXC 2050/1 - Project ID 390696704 - Cluster of Excellence "Centre for Tactile Internet with Human-in-the-Loop" (CeTI) of Technische Universität Dresden, and by the Federal Ministry of Education and Research of Germany in the programme of "Souverän. Digital. Vernetzt.". Joint project 6G-life, Project ID: 16KISK001K.

Author contributions Sebastian Ebert. and Johannes Mey. developed the main concepts and wrote most of the main manuscript text and prepared the figures. Measurements were conducted by Sebastian Ebert. The implementation was done by Sebastian Ebert with contributions from Johannes Mey. and René Schöne. Sebastian Götz. wrote parts of the abstract, introduction and conclusion sections. Uwe Aßmann. and Sebastian Götz. and R.S. contributed to the concepts and design decisions. All authors reviewed the manuscript.

Funding Open Access funding enabled and organized by Projekt DEAL.

Data availability No datasets were generated or analysed during the current study.

Declarations

Conflict of interest The authors declare no competing interests.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

1. Ciccozzi F, Di Ruscio D, Malavolta I, Pelliccione P, Tumova J (2017) Engineering the software of robotic systems. In: 2017 IEEE/ACM 39th International conference on software engineering companion (ICSE-C), pp 507–508. IEEE
2. Quigley M, Conley K, Gerkey B, Faust J, Foote T, Leibs J et al (2009) ROS: an open-source Robot Operating System. In: ICRA Workshop on open source software, vol 3, p 5. Kobe, Japan
3. Lesire C, Pommereau F (2018) ASPiC: an acting system based on skill petri net composition. In: International conference on intelligent robots and systems (IROS), pp. 6952–6958. <https://doi.org/10.1109/IROS.2018.8594328>. IEEE
4. Dondrup C, Papaioannou I, Lemon O (2019) Petri Net machines for human-agent interaction. <https://doi.org/10.48550/arXiv.1909.06174>
5. Pelletier B, Lesire C, Grand C, Doose D, Rognant M (2023) Predictive runtime verification of skill-based robotic systems using Petri Nets. In: 2023 IEEE International conference on robotics and automation (ICRA), pp 10580–10586. IEEE

6. Santos PMP (2016) PN-RTE, petri net robot task execution. Master's thesis, Técnico Lisboa
7. Figat M, Zieliński C (2022) Synthesis of robotic system controllers using robotic system specification language. *IEEE Robot Autom Lett* 8(2):688–695
8. Dal Zilio S, Hladik P-E, Ingrand F, Mallet A (2023) A formal toolchain for offline and run-time verification of robotic systems. *Robot Auton Syst* 159:104301
9. Halder R, Proença J, Macedo N, Santos A (2017) Formal verification of ROS-based robotic applications using timed-automata. In: 2017 IEEE/ACM 5th International FME workshop on formal methods in software engineering (FormaliSE). <https://doi.org/10.1109/FormaliSE.2017.9>. IEEE
10. Wang R, Guan Y, Song H, Li X, Li X, Shi Z, Song X (2018) A formal model-based design method for robotic systems. *IEEE Syst J* 13(1):1096–1107. <https://doi.org/10.1109/JSYST.2018.2867285>
11. Cheng BH, Clark RJ, Fleck JE, Langford MA, et al.: (2020) AC-ROS: assurance case driven adaptation for the robot operating system. In: Proceedings of the 23rd ACM/IEEE international conference on model driven engineering languages and systems. <https://doi.org/10.1145/3365438.3410952>
12. Kortik S, Shastha TK (2021) Formal verification of ROS based systems using a linear logic theorem prover. In: International conference on robotics and automation (ICRA), pp 9368–9374. <https://doi.org/10.1109/ICRA48506.2021.9561191>. IEEE
13. Zander S, Heppner G, Neugschwandner G, Awad R, Essinger M, Ahmed N (2015) A model-driven engineering approach for ROS using ontological semantics. In: 6th International workshop on domain-specific languages and models for robotic systems (DSLRob-15). <https://doi.org/10.48550/arXiv.1601.03998>
14. Estévez E, García A, García J, Ortega J (2018) ART²ool: a model-driven framework to generate target code for robot handling tasks. *Int J Adv Manuf Technol*. <https://doi.org/10.1007/s00170-018-1976-z>
15. Chaudhuri SR, Banerjee A, Swaminathan N, Choppella V, Pal A, Balamurali P (2019) A knowledge centric approach to conceptualizing robotic solutions. In: Proceedings of the 12th innovations on software engineering conference, pp 1–11. <https://doi.org/10.1145/3299771.3299782>
16. Kilgo P, Syriani E, Anderson M (2012) A visual modeling language for RDIS and ROS nodes using AToM 3. *Lecture notes in computer science* 7628 LNAI, 125–136 https://doi.org/10.1007/978-3-642-34327-8_14
17. Beaulieu A, Givigi SN, Ouellet D, Turner JT (2018) Model-driven development architectures to solve complex autonomous robotics problems. *IEEE Syst J* 12(2):1404–1413. <https://doi.org/10.1109/JSYST.2016.2583403>
18. Brugali D, Gherardi L (2016) HyperFlex: a model driven toolchain for designing and configuring software control systems for autonomous robots. *Stud Comput Intell* 625 https://doi.org/10.1007/978-3-319-26054-9_20
19. El Baccouri H, Guillou G, Babau J-P (2018) Robotic system testing with AMSA framework. In: MoDELS (Workshops), pp 316–325
20. Ramaswamy A, Monsuez B, Tapus A (2014) Saferobots: A model-driven approach for designing robotic software architectures. In: International conference on collaboration technologies and systems <https://doi.org/10.1109/CTS.2014.6867554>. IEEE
21. Baumgartl J, Buchmann T, Henrich D, Westfechtel B (2013) Towards easy robot programming—using DSLs, code generators and software product Lines. In: Proceedings of the 8th International joint conference on software technologies - volume 1: ICSoft-PT, (ICSOFT 2013), pp 548–554
22. Heinzemann C, Lange R (2018) vTSL—a formally verifiable DSL for specifying robot tasks. In: IEEE/RSJ International conference on intelligent robots and systems (IROS), pp 8308–8314. <https://doi.org/10.1109/IROS.2018.8593559>
23. Bencomo N, G"otz S, Song H, (2019) Models@run.time: a guided tour of the state of the art and research challenges. *Int J Softw Syst Model* <https://doi.org/10.1007/s10270-018-00712-x>
24. Ebert S, Mey J, Schöne R, Götz S, Aßmann U (2023) DiNeROS: A model-driven framework for verifiable ros applications with Petri Nets. In: 2023 ACM/IEEE International conference on model driven engineering languages and systems companion (MODELS-C), pp 791–800. IEEE
25. Reisig W (2012) Petri Nets: an introduction vol. 4. Springer, Heidelberg. <https://doi.org/10.1007/978-3-642-69968-9>
26. Peterson JL (1977) Petri Nets. *ACM Comput Surveys (CSUR)* 9(3):223–252. <https://doi.org/10.1145/356698.356702>
27. Hillah LM, Kindler E, Kordon F, Petrucci L, Trèves N (2009) A primer on the Petri Net Markup Language and ISO/IEC 15909–2. *Petri Net Newsletter* 76:9–28
28. Jensen K (1983) High-level Petri nets. In: applications and theory of Petri Nets: selected papers from the 3rd European workshop on applications and theory of Petri Nets Varenna, Italy, September 27–30, 1982 (under Auspices of AFCET, AICA, GI, and EATCS), pp 166–180. https://doi.org/10.1007/978-3-642-69028-0_12. Springer
29. Berthomieu B, Vernadat F (2006) Time petri nets analysis with TINA. In: Proceedings of the 3rd international conference on the quantitative evaluation of systems, vol 6, pp 123–124. <https://doi.org/10.1109/QEST.2006.56>
30. Rosjava. Accessed: 2023-01-30 (2017). <http://wiki.ros.org/rosjava>
31. Behrmann G, David A, Larsen KG (2004) A tutorial on Uppaal. Formal methods for the design of real-time systems, 200–236 https://doi.org/10.1007/978-3-540-30080-9_7
32. Holzmann GJ (2004) The SPIN model checker: primer and reference manual vol 1003. Addison-Wesley, Reading
33. Luckcuck M, Farrell M, Dennis LA, Dixon C, Fisher M (2019) Formal specification and verification of autonomous robotic systems: a survey. *ACM Comput Surveys* 52(5):1–41. <https://doi.org/10.1145/3342355>
34. de Araújo Silva E, Valentin E, Carvalho JRH, da Silva Barreto R (2021) A survey of model driven engineering in robotics. *J Comput Lang* 62:1021. <https://doi.org/10.1016/j.cola.2020.101021>
35. Moutinho F, Gomes L (2014) Asynchronous-channels within Petri net-based GALS distributed embedded systems modeling. *Trans Ind Inf* 10(4):2024–2033. <https://doi.org/10.1109/TII.2014.2341933>
36. Bera D et al.: (2014) Petri nets for modeling robots. PhD thesis, Einhofen University of Technology
37. Milutinovic D, Lima P (2002) Petri net models of robotic tasks. In: Proceedings 2002 IEEE international conference on robotics and automation, vol 4, pp 4059–4064. <https://doi.org/10.1109/ROBOT.2002.1014376>
38. Kotb YT, Beauchemin SS, Barron JL (2007) Petri net-based cooperation in multi-agent systems. In: Fourth Canadian conference on computer and robot vision (CRV), pp 123–130. <https://doi.org/10.1109/CRV.2007.49>. IEEE
39. Hedin G, Magnusson E (2003) JastAdd—an aspect-oriented compiler construction system. *Sci. Comput. Progr.* 47(1):37–58
40. Hillah L-M, Kordon F, Petrucci L, Trèves N (2010) PNML framework: an extendable reference implementation of the Petri Net Markup Language. In: 31st International conference on applications and theory of petri nets, Braga, Portugal. Springer
41. Almeida PS (1997) Balloon types: controlling sharing of state in data types. In: ECOOP'97—11th European conference object-oriented programming Jyväskylä, Finland, pp 32–59. <https://doi.org/10.1007/BFb0053373>. Springer
42. Jensen K (1996) Coloured petri nets: basic concepts, analysis methods and practical use. Springer, Heidelberg. <https://doi.org/10.1007/978-3-662-03241-1>

43. Schöne R, Mey J, Ebert S, Götz S, Aßmann U (2022) Incremental causal connection for self-adaptive systems based on relational reference attribute grammars. In: Proceedings of the 25th international conference on model driven engineering languages and systems, pp 1–12. <https://doi.org/10.1145/3550355.3552460>
44. Minas M, Frey G (2002) Visual PLC-programming using signal interpreted Petri nets. In: Proceedings of the American control conference, vol 6, pp 5019–5024. <https://doi.org/10.1109/ACC.2002.1025461>. IEEE
45. Vyatkin V, Hanisch H (2000) Practice of modeling and verification of distributed controllers using signal net systems. In: International workshop on concurrency, specification and programming
46. Berthomieu B, Le Botlan D, Dal Zilio S (2020) Counting Petri net markings from reduction equations. *Int J Softw Tools Technol Transfer* 22:163–181

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.