Modeling and Verification of a Protocol for Operational Support using Coloured Petri Nets

Michael Westergaard⋆ and Fabrizio M. Maggi⋆⋆

Department of Mathematics and Computer Science,
Eindhoven University of Technology, The Netherlands
{m.westergaard,f.m.maggi}@tue.nl

Abstract. In this paper, we describe the modeling and analysis of a protocol for operational support during workflow enactment. Operational support provides online replies to questions such as "is my execution valid?" and "how do I end the execution in the fastest/cheapest way?", and may be invoked multiple times for each execution.
Multiple applications (operational support providers) may be able to answer such questions, so a protocol supporting this should be able to handle multiple providers, maintain data between queries about the same execution, and discard information when it is no longer needed.
We present a coloured Petri net model of a protocol satisfying our requirements. The model is used both to make our requirements clear by building a model-based prototype before implementation and to verify that the devised protocol is correct.
We present techniques to make analysis of the large state-space of the model possible, including modeling techniques and an improved state representation for coloured Petri nets allowing explicit representation of state spaces with more than 10^8 states on a normal PC.
We briefly describe our implementation in the process mining tool ProM and how we have used it to improve an existing provider.

1 Introduction

In business process management, *operational support* [10] is the capacity to provide users with recommendations about actions to be taken in order to arrive at a goal. In this paper, we aim at extending an existing infrastructure for operational support to allow stateful communication between a client and a set of operational support providers. We specify our design using a coloured Petri net model to define the protocol and to get a firm idea of our requirements. Afterward, we use state space analysis to verify that the devised protocol is correct. Finally, we use the model as blueprint for implementation.
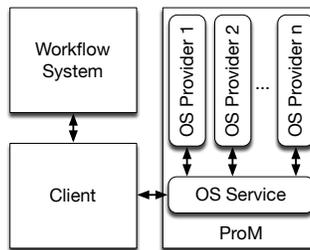
Operational support allows a number of queries: *simple*, *compare*, *predict*, and *recommend*. A client sends a query and a partial execution trace of a workflow, and gets a response to the query. In this paper we abstract away the actual contents of the queries, so suffice to say that these four types represent increasingly complex queries, where the simplest kind allows clients to ask diagnostics about the current execution (such as whether the current execution is valid) and the more complex queries allow clients to ask for an optimal strategy for a desired goal (such as completing an execution as fast as possible).

An infrastructure for operational support is already implemented in the process mining tool ProM [9]. It is shown in Fig. 1. A Client communicates with a Workflow System and with the *operational support service* (OS Service; OSS in the following). The OSS forwards requests to a number of *operational support providers* (OS Providers; providers in the following), which may implement different algorithms, and sends back replies.

The major problem with this implementation is that if a client provides the OSS with a trace

Fig. 1. Existing operational support architecture.

and later with an extension, it is often possible to reuse a lot of the first computation, but as the current implementation is state-less, this is not possible. This can be addressed by making the OSS stateful, so the partial trace and results of any computation performed on it can be stored in a session. Statefulness can be implemented by each provider individually but this would be doubling a lot of effort, and we instead propose to let the OSS handle sessions for all providers, including session management such as serializing sessions on service shutdown and session garbage collection on client shutdown. Our aim is to put all complexity of session handling inside the OSS, so clients and providers remain simple (as we have more of them but only one OSS).

Our initial specification for the new operational support was as vague as "operational support with sessions". Rather than starting an implementation from this vague definition or making a more detailed textual specification, we decided to make an executable specification as a coloured Petri net (CPN) [6] model. The resulting CPN has been useful, both as a specification and as an executable prototype for verification of correctness of the devised protocol. After constructing the model, we had to address the problem that the state-space of the model was very large (exhausted memory around $10^5 - 10^6$ states). Model-alterations and implementation of a more efficient state representation allowed us to analyze the model with $10^7 - 10^8$ states, enough for this case. Modeling revealed several under-specified parts and verification revealed further problems (memory leaks and dead-locks), which were found and fixed before implementation started.

The contribution of this paper is three-fold: first, we present a viable protocol for operational support with sessions, second, we demonstrate that coloured Petri nets can be used for both making an abstract idea of a protocol more concrete and for verifying that the final design is correct even though the resulting state-

space is very large, and, third, we present a very efficient state representation for CPNs. We have spent around 1 man week on the construction of the model and around 2 man weeks on implementing a new state space analyzer.

The rest of this paper is structured as follows: in the next section, we provide the background needed to understand the rest of this paper. In Sect. 3, we present the developed model, and in Sect. 4, we sum up the analysis phase and the required fixes to the model. In Sect. 5, we outline an implementation of the devised protocol and its advantages for a real-life provider. Finally, in Sect. 6, we draw our conclusions and provide directions for future work. The reader is assumed to have a basic knowledge of coloured Petri nets but no prior knowledge of operational support is assumed.

2 Background

In this section we briefly introduce coloured Petri nets (CPNs) and the old operational support service (OSS). CPN is a high-level Petri net formalism, extending standard Petri nets. CPNs are bipartite directed graphs comprising *places*, *transition*s and *arc*s, with inscriptions and types allowing tokens to be distinguishable. In Fig. 6 we see (part of) a CPN. Places are ovals (e.g., Working) and transitions are rectangles (e.g., Check). Places have a *type* (e.g., CLIENTxSESSIONID) and can have an *initial marking* which is a multi-set of values (tokens) of the corresponding type. Arcs contain *expressions* with zero or more free *variables*.

We call a transition and assignment of values to all its free variables a *binding element* or *binding* (e.g., Check with c=1, sid=2, t1=[] and trace=[] written Check⟨c=1,sid=2,t1=[],trace=[]⟩). An arc expression can be evaluated in a binding using the assignments, resulting in a value (e.g., (1,2) on the arc from Working to Check in the previous binding). A binding is *enabled* if all input places contain at least the tokens prescribed by evaluating the arc expressions (in the example no binding elements are enabled; if Working contained a token (1,2) and Trace a token (1, ([],[])) the binding Check⟨c=1,sid=2,t1=[],trace=[]⟩ would be). An enabled binding can *occur*, removing the corresponding tokens on input places and creating new tokens on output places (in the example, the binding would among others remove the token (1,2) from Working and move it to Waiting).

CPNs can contain multiple *module*s (or *pages*). The interface of a module is described using port places, places with an annotation In, Out, or I/O (e.g., Working is a port place and Waiting is not). A module can be represented using a *substitution transition*, which is a rectangle with a double outline (e.g., Query in Fig. 5). Places connected to a substitution transition are called *socket places* and are connected to port places using *port/socket assignment*s. We use the convention that places in a port/socket assignment share the same name.

2.1 Operational Support Service

We have already introduced the basic operation of the old operational support service (OSS) in Sect. 1, so instead of repeating that, we focus on a single

provider, the Declare Monitor, and stress some of the problems that the old OSS encountered in this provider. In Sect. 5, we show how these issues have been addressed in the new implementation.

Monitoring Declare Models. The Declare Monitor takes as input a Declare model [8] consisting of events and constraints on the events. In Fig. 2 we see an example Declare model of a selling process. Here ReceivePayment must be directly followed by SendInvoice (expressed by the constraint chain_response) and ReceiveOrder must be eventually followed by ArchiveOrder (response).
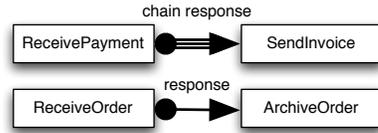

Fig. 2. Example Declare model

The provider receives partial traces from clients (e.g., different executions of the selling process) and monitors the behavior of these w.r.t. the Declare model. When an event occurs, the Declare Monitor associates to each constraint of the Declare model one of the states: *satisfied*, *pending*, or *violated*. A constraint is satisfied if it is currently not violated. The pending state indicates that a constraint is currently violated but can become satisfied later. This is e.g. the case for constraints waiting for a specific event. Finally, a constraint is violated if it is currently violated and can never become satisfied again.

In Fig. 3, a client for the Declare Monitor is shown. When event ReceiveOrder initially occurs, response becomes pending (waiting for the event ArchiveOrder). ArchiveOrder never occurs in the trace so, when the execution completes, the constraint is violated. The constraint chain_response is initially satisfied. When event ReceivePayment occurs, it becomes pending (waiting for SendInvoice). When SendInvoice occurs (immediately after ReceivePayment) the constraints is satisfied again. When event ReceivePayment occurs again, it is not followed directly by SendInvoice and the constraint is violated.

The first issue identified in the existing OSS is that when a client sends a request, the OSS unconditionally sends the request to all providers, regardless of whether they can handle it or not. Second, any information needed by the provider to perform analysis must be specified when the provider is started, so clients cannot customize a provider. For instance, in the Declare Monitor, all the clients use the Declare model specified when the provider is started, so a single server cannot monitor different processes. Third, the old protocol is state-less, so a provider receives all information related to a client in each request. Therefore,
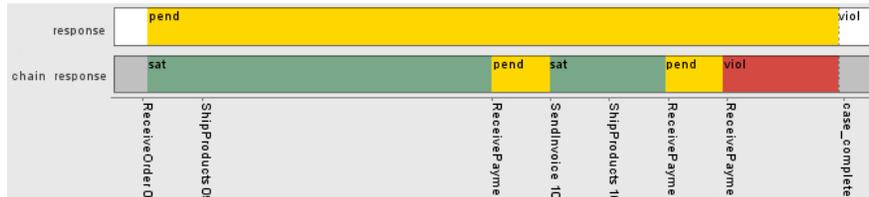
Fig. 3. Declare Monitor client

the Declare monitor is forced to check the entire partial trace statically every time, making run-time monitoring impossible (and impractical to simulate as the start-up cost of the Declare monitor is very large). Finally, clients cannot inform providers that an execution is completed. For the Declare Monitor, this information is crucial to report pending constraints as violated when execution ends.

3 Model of the New Protocol

We wish to extend the existing operational support service (OSS) so it supports sessions with persistent data between calls from the same client instance. We want to preserve a loose coupling between client and provider, but we still want to be able to make an intelligent pairing, so a client instance and a provider are only paired if the provider is able to provide meaningful responses to queries from the client.

The basic idea of the protocol is that a client sends a CREATE_SESSION message to the OSS. The OSS queries all available providers if they wish to be attached to the session, and the OSS responds back to the client with a SESSION_CREATED message. The client now sends CHECK messages connected to a session, which are distributed to all providers attached to the session, and an aggregate result is returned to the client in a CHECKED message. A client can send a CLOSE_SESSION message to the OSS, which is distributed to all providers registered with the session, and finally a SESSION_CLOSED message is returned to the client. Providers may register themselves with the OSS by a REGISTER_PROVIDER message and deregister themselves with a SHUTDOWN_PROVIDER message. This can happen at any time.

Our focus with this model is on the correctness of the protocol with respect to session management, so we have abstracted away any data not directly related to this and made any decision based on data using a non-deterministic choice. We have build a more detailed model than explained in this paper (explicitly modeling data and also a backwards compatibility layer). We can switch features on and off to reflect the different uses of the model (specification and verification) and for simplicity have chosen to just explain the model used for verification here. We have chosen to model all processes as terminating (so, for example, once a client instance has terminated, it will never again become alive to send requests). The model is rather large and consists of 25 modules and a total of 134 places and 65 transitions.

The top level of our model is shown in Fig. 4. We have clients to the left and the operational support server to the right (the server comprises the OSS and all providers). They communicate via two places, one for sending requests from the client to the server and one for getting replies from the server. Requests
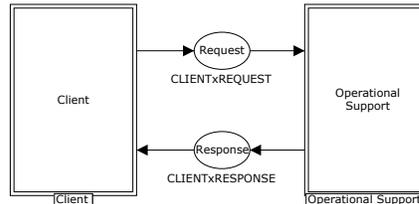
Fig. 4. Operational support model.

are tied to a specific client, effectively modeling a bidirectional channel between each client and the server. We assume that no communication channels lose packets. For most channels we allow participants to consume any transmitted packet in arbitrary order (more on this later). Our focus is on the operational support server, but we briefly introduce the client part to get a better understanding of the environment of the server.

3.1 Client

The model of the clients is shown in Fig. 5. We have folded all clients into a single net, so all places have a CLIENT component identifying the client. Clients are assumed to be single-threaded (or at least synchronize around communication with the server, so each has at most one outstanding message). Clients start in the Idle state and can Start Session by sending a CREATE_SESSION message to the server, receiving a SESSION_CREATED message back and transitioning to the Working state. In the Working state, a client can either Execute tasks internally or it can perform a Query to the operational support server. Executing tasks naturally updates our local view of the execution Trace, and sending a Query accesses the Trace as well as the communication channels to the server. When we Close Session, aside from notifying the server, we can either go to the Off-line state or the Done state. In the Off-line state we can continue Execute tasks, but we can no longer make queries. From the Off-line state we can Start Session anew and get back to the Working state. In the Done state, the client does no more work, and all messages from the server are just Discarded.
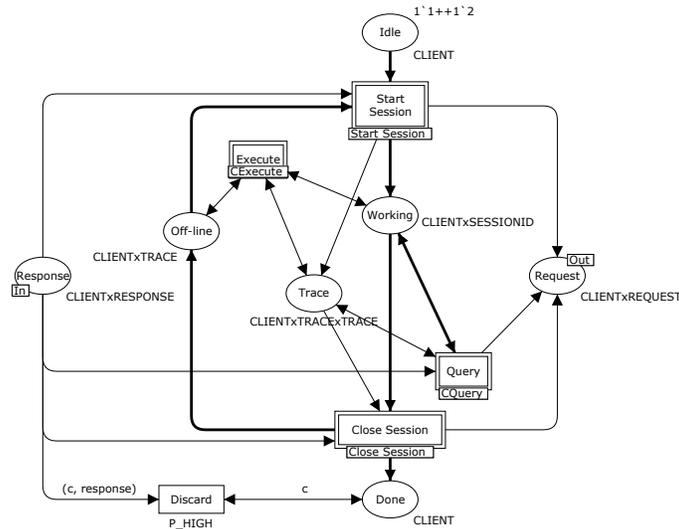

Fig. 5. Client.

We have modeled all types
of queries as a single action
Check in the query module
(Fig. 6). The Check transi-
tion reads the Trace and trans-
mits a CHECK message to the
server, containing the id of
the session, the part of the
trace currently not sent to the
server, and a query (here just
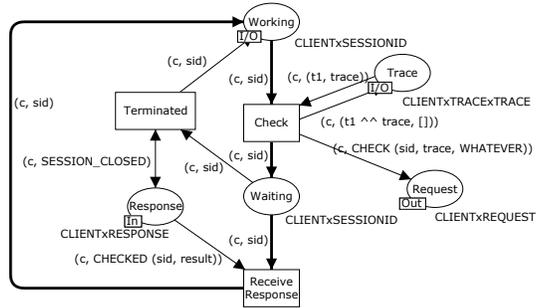modeled as the value WHAT-
EVER as we are not interested

Fig. 6. Query module of client.

in the actual contents but want to model that information is sent). The Trace
actually contains two traces: one for events that have not yet been seen by the
server and one for events that have already been seen by the server. When we
send a message to the server, we move all events from the first to the latter
and only send new events. After executing a query, we go to the Waiting state
and wait for response. From this state, we can Receive Response and go back
to the Working state if we receive a CHECKED message. We may also receive a
SESSION_CLOSED message from the server if the server has decided to garbage
collect the session. Here we also go back to the Working state so we can share
the session shutdown (and optional revival) procedure with the successful case.

3.2   Operational Support Server

The operational support server (Fig. 7) consists of an Operational Support Service
(OSS) receiving messages from the client via Request. Requests are handled and
optionally forwarded to one or more Operational Support Providers via Provider
Request. Providers send back responses via Provider Response, and the OSS com-
bines responses corresponding to a request and sends a Response back to the
client containing the aggregated results from all providers. The OSS makes sure
to instantiate and remove Sessions but providers can both read from and write
data to sessions. A session is essentially a representation of the executed trace
(the OSS makes sure this matches the values of the place Trace of Fig. 12 to the
best of its knowledge) and a key/value mapping, allowing providers to store any
named data.

3.3   Operational Support Service

The Operational Support Service (OSS) module is by far the most complicated.
This reflects the desire to put as much functionality (and hence complexity) as
possible inside this module to keep the clients and providers simple. Furthermore,
we have tried to make locking as fine-grained as possible to increase concurrency.
The OSS (Fig. 8) consists of two logical parts: handling communication from the
client (the part of the figure to the left of places Provider Request and Provider
Response) and communication from the providers (the part to the right).

The simplest part is the provider side, which handles new providers and registers them (Register Provider) as well as shuts them down (Shutdown Provider). Together these modules take care of maintaining a view of all registered providers on AllProviders. These receive input on the Provider Response channel, which is the channel used for communication from the provider to the OSS. Neither of these sends responses back to the providers as we do not expect a provider to be able to receive messages after announcing a shutdown and providers do not need any response after setup (as we assume that packets are not lost).

The client handling part performs Session Handling and handles Queries from clients. Both of these activities involve receiving messages on Request, passing them through to Provider Request, receiving responses on Provider Response and passing them on to Response. Session Handling maintains all Sessions according to requests from clients and garbage collection rules. For analysis reasons, we only allow a limited number of sessions to be created, and the number of Available Sessions keeps track of how many extra sessions we can create. We furthermore maintain a database of all currently active session ids on AllSessions (which is
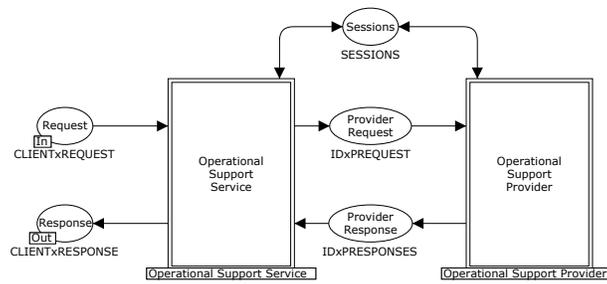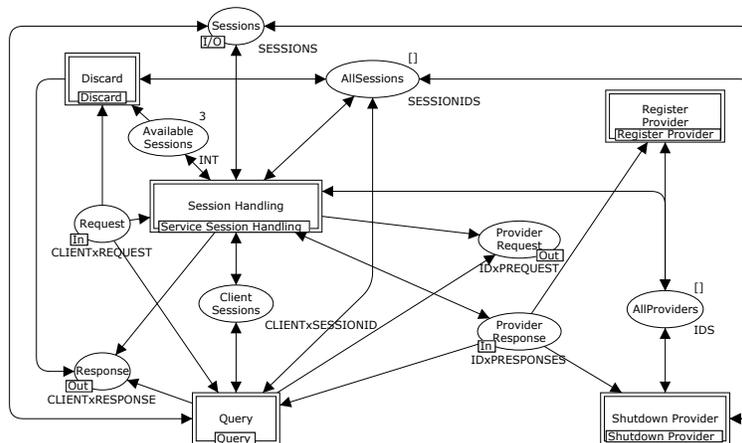

Fig. 7. Server.

Fig. 8. Operational support service (OSS).

used to allow modules to detect when garbage collection has taken place in the middle of processing) and a mapping from client ids to session ids on Client Sessions, which is used to make sure that clients cannot hijack the session of another client. The Query module receives queries and passes them on to the correct recipient providers. The client handling part also has a Discard module, which takes care of requests from clients after we have used up all Available Sessions, and just emulates a session that is created upon request and immediately garbage collected. This makes sure that clients do not deadlock after the artificially imposed bound on the number of sessions that can be created has been reached, and would not exist in an implementation.

Session Handling. Session handling consists of three parts: session setup (Fig. 9), session tear-down (Fig. 10), and garbage collection (Fig. 11). Session setup is triggered when a CREATE_SESSION message is received. Session setup shares the layout with all main OSS modules, namely we have input from/output to the client to the left and output to/input from providers to the right, and state moves from the top to the bottom. Create Session generates a new session id (sid) using Available Sessions and adds the session to AllSessions and Client Sessions for later use. We generate a new empty session on Sessions. A session is a tuple consisting of a session id, a pair of new and old execution traces (initially both empty), and a mapping from provider id to a set of key/value pairs, i.e., a data storage for each provider. We get all providers from AllProviders. We furthermore inform all providers that the session has been created, and move to the Pending Setup state noting the client, session id, and providers from which to expect a response. In the Pending Setup state we receive positive or nega-
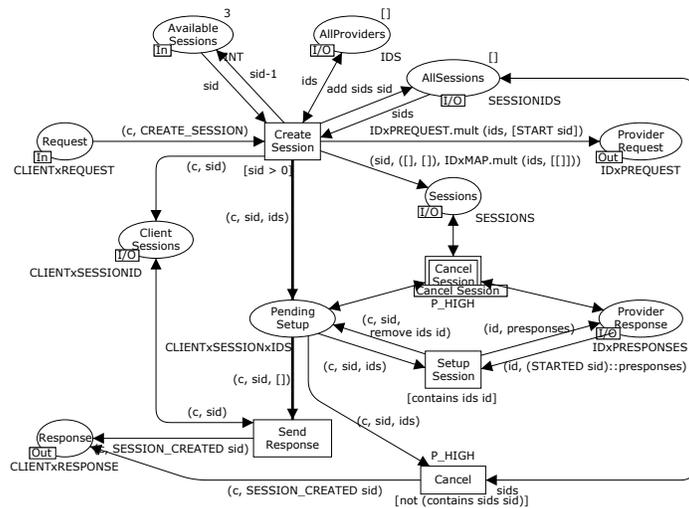

Fig. 9. Session setup module.

tive acknowledgments from all providers associated with a session. If a provider sends a STARTED message, we successfully Setup Session for that provider, and remove it from the list of providers for which we expect a reply. If a provider rejects the session using a SESSION_ENDED message or has shut down, sending a SHUTDOWN_PROVIDER message, the session is canceled, and the entry in the session containing information about that provider is removed. The details are hidden in a module so as not to clutter the model unnecessarily. The Provider Response channel is the only channel modeled as an ordered channel to avoid session response messages building up in the channel if a provider first accepts a session and then shuts down. When the list of providers we expect a response from is empty, Send Response becomes enabled and sends a SESSION_CREATED message back to the client. The Cancel transition aborts session setup if the session has been garbage collected while waiting for response from providers. Even if we cancel a session, we send a SESSION_CREATED message to the client, so the client does not have to handle session shutdown during setup, but only during queries.

Session tear-down (Fig. 10) is handled similarly. When a CLOSE_SESSION message is received, it is matched with Client Sessions, the session trace information is updated, all providers are notified, and the OSS transitions to the Waiting state, noting all providers from which it expects a response. If a provider has been shutdown, we Discard the entry for the provider and if we Receive Confirmation we do the same. When we no longer wait for response from any providers, we remove the session from Sessions, AllSessions and Client Sessions, and send SESSION_CLOSED back to the client. As before, we can Cancel the operation if the session has been garbage collected while waiting for response.

Session garbage collection (GC) (Fig. 11) is modeled in a simple way: we GC a session if it has no more registered providers. We could make this more elaborate, e.g., allowing GC to take place if a session has a certain age.
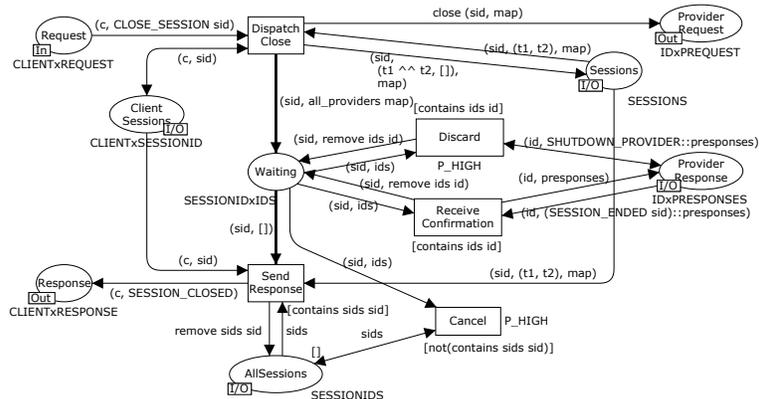

Fig. 10. Session tear-down module.

Query. Queries are handled (Fig. 12) similarly to session handling: when the OSS receives a query message (CHECK) it checks the validity in Client Sessions, updates the trace in Sessions and dispatches to all providers. In the Waiting state it is possible to Gather Results, storing the aggregate result in Intermediate Results. Here the aggregation is just

Fig. 11. Session GC module.

boolean or between truth values returned by providers, (in fact, for analysis, we configure the model so the produced response is always false and no aggregation actually takes place). If a provider has terminated the connection, we stop waiting for a response and Discard Request. We can also Cancel processing if the session has been garbage collected. When we have received a response from all providers, we Send Response back to the client.

Provider Handling. Provider handling is responsible for maintaining the AllProviders view of active providers as well as for removing provider-specific data from all sessions when a provider shuts down. When a new provider registers itself using a REGISTER_PROVIDER message (Fig. 13) it is added to AllProviders. When a provider shuts down, sending a SHUTDOWN_PROVIDER (Fig. 14), it is removed from AllProviders and all provider-specific data is scheduled to be

Fig. 12. Query module.

removed from all sessions. As long as a session containing provider-specific data for the removed provider exists, the provider-specific data is Removed. We note that at no point do we assume that we have exclusive access to all sessions at once here. When no more sessions registered with the provider are available (a session may

Fig. 13. Register provider module.

be shut down between the shutdown of the provider and the removal of provider-specific data from that session), the request is Done.
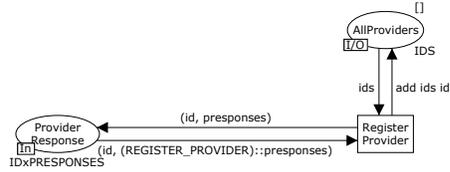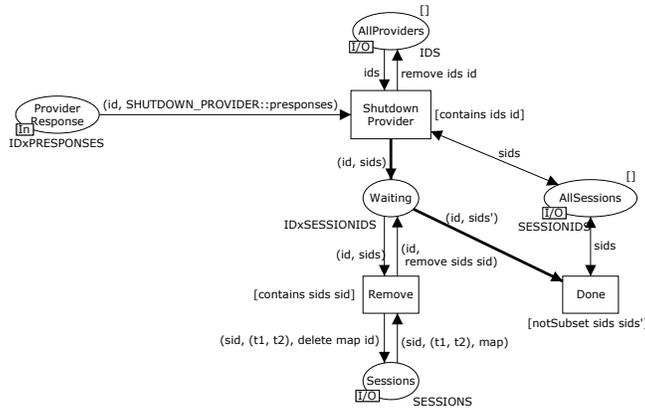

Fig. 14. Shutdown provider module.

3.4 Operational Support Provider

A provider (Fig. 15) performs two administrative tasks, Provider Registration and Session Handling, and one actual task, handling Queries. Any request to a Shut Down provider is Discarded.

The provider registration module (Fig. 16) implements a simple life-cycle for providers: they are initially Stopped and after calling Register Provider and sending a REGISTER_PROVIDER message to the

Fig. 15. Provider module.

OSS, they are Started. Finally, they can call Remove Provider to transition to the Shut Down state and send a SHUTDOWN_PROVIDER message to the OSS.
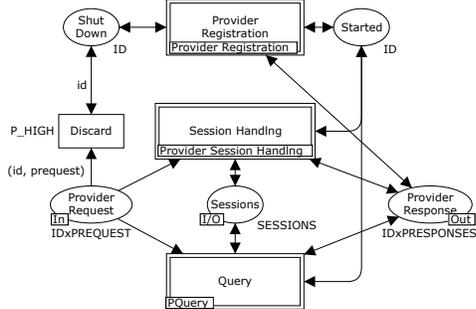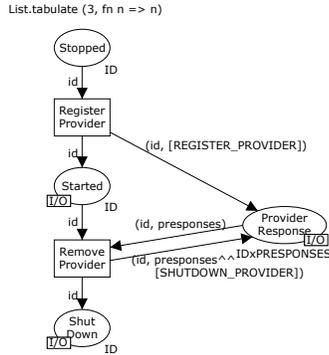
Fig. 16. Provider registration module.

Session handling for providers is simple. It consists of two parts, session setup (Fig. 17) and ending sessions (Fig. 18). Setup is handled when a provider receives a START message from the OSS. If a provider is Started, it can choose to Setup Session and respond with a STARTED message. In the model, we also store the provider name in the provider-specific data of the Session. If a provider is no longer started (i.e., if it has shut down after the OSS sent the message but before it was handled), it chooses to Reject Session and reply with a SESSION_ENDED message. In fact, when a provider is Started, a non-deterministic choice is made between Setup Session and Reject Session to simulate that a provider may be incompatible with the services requested in the session. An implementation would of course not make a non-deterministic choice here, but inspect meta-data in the session request and make an informed decision. End Session (Fig. 18) is invoked when an END_SESSION message is received and sends back a SESSION_ENDED message. On termination a provider has access to the data of the session, and can use that to shut down resources referred to in the session.

Queries are handled (Fig. 19) in much the same way: When a PCHECK message is received, we access the Sessions to get provider-specific data, make sure that the provider is actually Started and send a PCHECKED message back to the OSS. The contents of the response is generated randomly as a boolean.
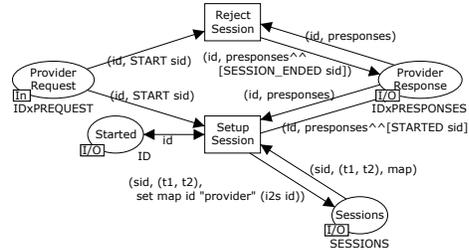
Fig. 17. Setup session module.

4 Analysis

The main goal of the model developed is to serve as specification of the operational support protocol. We have tested the protocol with all features enabled using simulation but also in an iterative process using state-space analysis with
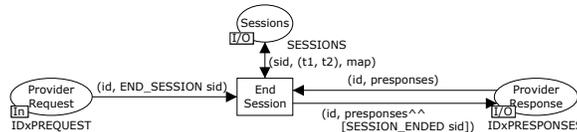
Fig. 18. End session module.

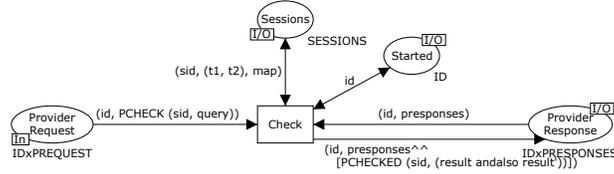**Fig. 19.** Query module of provider.

some features disabled. State-space analysis is a common technique for analysis of formally modeled systems as it promises fully automatic and complete coverage of all behavior of the modeled system. The method suffers from the state explosion problem, namely that the size of the state-space may be exponential or even larger in the size of the model. In this case, we also encounter the state explosion problem. We describe some techniques we have used to reduce the size of the state-space by means of model alterations, use of transition priorities, and a very efficient state-representation for general CPNs. Furthermore, we describe the errors found during analysis and how we have fixed them.

Model Alterations. Model alterations can drastically reduce the size of the state-space. In our model, the client is allowed to execute arbitrary events an arbitrary number of times (Fig. 5), causing, among others, the Trace place to be unbounded and hence the state-space to be infinite. For analysis, we therefore switch off the ability to execute events. Related to this, we have also abstracted away most data not having anything directly to do with session handling, and we have restricted the number of available sessions in Fig. 9.

Aside from standard data-abstraction, we have also used transition priorities to reduce the state-space. The idea is that sometimes tokens are produced that will have no effect on the future execution. One such example is packets that should be discarded, such as packets sent to a shut down provider in Fig. 15. Instead of non-deterministically discarding the tokens, we discard them immediately. Otherwise, we basically double the size of the state-space for each such possible token, as we have one copy of each following marking where the token is discarded immediately and one where it is discarded last (and the two copies are interconnected by discarding the token at any intermediate point). By discarding the token immediately we only have one copy, reducing the state-space. We do the same reduction for the Cancel transitions in Figs. 9, 10 and 12, and for Discard Session in Fig. 11.
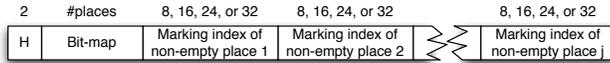
New State-space Tool. We used exploratory modeling to get a hold of our requirements for the protocol. This involved using state-space exploration as a debugging technique. In order to accommodate that, we used two known techniques: translating liveness-properties to safety properties (really dead-locks) and

using depth-first traversal to find errors as fast as possible and report them on-the-fly. To support that, we could not easily use the built-in state-space generator in CPN Tools [3], as it generates the state-space in a breath-first manner and is designed around off-line verification. We could also not directly use ASAP [11] as it has no support for priorities of transitions.

Safety properties, including dead-lock freeness and bound violations, are a particularly simple kind of properties that can be determined by looking at each state in isolation. Not all interesting properties are safety properties, though. This includes a property like "any session created is eventually torn down if the client terminates", which can only be determined by looking at all (possible infinitely many) execution traces. This can be done on-the-fly but requires representing the synchronized product of the state-space and a property automaton, making it infeasible in our case. Instead we translate such interesting properties to safety properties by making sure that no entities (clients, sessions, and providers) in the model can be revived after terminating. This ensures the system has goal states (all clients and providers have shut down) and we can check if the system has dead states where not all sessions have been torn down.

As we could not use any off-the-shelf state-space generator, we made our own. We did not have complicated requirements as we only had to find dead states and provide error-traces. For this reason we decided to use Access/CPN [12], which makes it possible to access the CPN simulator used in CPN Tools from Java. The reason for not implementing this in ASAP instead is that Java compared to Standard ML (the language used in ASAP) allows finer grained control over memory, access to more memory, and support for threads.

During initial analysis, we noticed that many places share the same marking and are often empty. This prompted us to implement an efficient state representation exploiting this. The final representation is shown in Fig. 20. CPN Tools already uses a state-representation sharing on the levels of values, markings, and places [2], but it is structured, requiring several pointers for each place. Instead we propose using an unstructured representation: store all multi-sets as strings and enumerate them. We can then store the marking of a place using an integer. Instead of storing places in a structured way, we enumerate them, allowing us to represent a state using $\#places \cdot |integer|$ bits (plus the overhead of storing the multi-sets). Rather than of wasting an integer for places with empty markings, we store a bit-map indicating non-empty places, and only store a marking for them. Finally, we do not need to use an entire machine word to store an integer. Instead, we can use just 8, 16, 24, or 32 bits depending on the value of the largest integer used. We do not need more bits as, if we have more than 2^{32} different multi-sets, chances are we are not able to analyze the model anyway. We add to each state two bits indicating the number of bytes used for each integer, resulting the representation shown in Fig. 20. We translate to this representation using a map from multi-sets to their enumerated value and a counter for the next multi-set id, inspecting places in a canonical order. We translate from this representation by additionally having an array storing each multi-set in the en-

2	#places	8, 16, 24, or 32	8, 16, 24, or 32			8, 16, 24, or 32
H	Bit-map	Marking index of non-empty place 1	Marking index of non-empty place 2	>	<	Marking index of non-empty place j

Fig. 20. Compact state representation.

try with its id. This representation is generic and has also proved efficient for other models.

We currently store each state in a separate array, imposing an overhead of 16 bytes plus enough to make it a multiple of 8 bytes. In our case, we used a total of 3.18 giga-bytes to represent 68,923,926 states (including overhead and 8.8 mega-bytes for mappings between multi-sets and ids), for an average of 49.5 bytes per state. To store these, we use an additional 2–4 pointers (8–16 bytes), for at total of up to 65.5 bytes per state, allowing us to store this in just over 4 giga-bytes of memory. We can represent more than 10^8 states explicitly in memory on a standard computer with 8 giga-bytes of memory (we can use 32-bits to address up to 32 giga-bytes of memory by using packed pointers addressing objects aligned at positions divisible by 8), which we have made use of on several intermediate models. Without reduction, CPN Tools was able to generate 200,000–300,000 states before exhausting available memory. The amount of memory used by our compact representation is comparable to other memory-efficient but potentially time-consuming explicit representations of the state-space for coloured Petri nets. For example, the ComBack [4] method uses 140–150 bytes per state for realistic examples in the Standard ML implementation. In theory the ComBack method uses just 20 bytes per state but does so by imposing a quite heavy penalty on execution time. Using the sweep-line method [7] it is possible to represent a state in around 8 bytes plus 8 bytes per transition at the cost of not easily being able to reconstruct a compactly represented state, a potential cost of re-exploring parts of the state-space, and a peak memory usage which may be much larger. These methods can also gain from our compact state representation for intermediate representations and caches. Other methods for compact state representation, such as bit-state hashing [5] and hash compaction [13], do not guarantee full coverage, and there is no good approach for symbolic representation of fully general CPNs as it is not known how to represent the transition relation symbolically.

4.1 Errors Found

The non-trivial parts of the protocol consist of the interaction between client communication and session garbage collection, and handling of provider termination. For example, a scenario where a client sets up a session, initiates a query, and during the query one of the providers handling the request terminates, should be handled by the OSS. As all entities in our model are terminating (i.e., cannot be revived after stopping), this corresponds to checking that in all dead states, all entities are in their shut down state (meaning they did not dead-lock)

and that in all dead states there are no sessions alive (so all sessions can be terminated correctly), no providers are registered (so the OSS does not dead-lock in an undesired state), and there are no outstanding messages on either of the communication channels (so neither of the components dead-locked or did not process all messages correctly).

We had several cycles of modeling and analysis, and after fixing some minor initial modeling problems, we were left with three major bugs. The first bug was in the interplay between OSS communication with providers and provider shutdown. The second bug was mainly due to the fact that we modeled all communication channels without an explicit ordering, and the third bug was more serious and caused by the interplay between session garbage collection and provider communication.

The first version of our model did not consider a provider terminating in the middle of a query or session tear-down, leading to dead-locks in the OSS (dead states where the OSS was in Waiting in Figs. 10 and 12). Instead the OSS would wait indefinitely for a SESSION_ENDED which would never arrive. We therefore added Discard transitions in those cases, solving this.

The second problem arose from the fact that the protocol between the provider and the OSS is one-way. If a provider replies to a query and immediately shuts down, the OSS would have both an answer and a SESSION_ENDED message for the same provider in Provider Response in Figs. 9, 10 and 12, enabling both the Discard/Cancel Session transition and the transition for successfully receiving a response (Setup Session/Receive Confirmation/Gather Results). We could fix this by adding an explicit response for each message to the provider, but this would unnecessarily clutter the model. As we expected to implement the protocol over TCP (which ensures in-order delivery of messages), we instead modeled this channel as first-in/first-out ordered, thereby fixing the problem. To keep the model simpler, we do not impose this on the remaining channels.

Initially, we did not have Cancel transitions on the OSS components in Figs. 9, 10 and 12. If a client made a request, registered one provider, made a query, and then the provider shut down (causing the session to be garbage collected), the OSS would be stuck in the Waiting state. This third problem can occur in various forms for all three modules, and is fixed by adding the AllSessions place maintaining a view of all sessions not garbage collected and using that information to Cancel a request if necessary. Our initial solution had the Cancel transitions explicitly notify clients that a request was canceled, but this could lead to stale messages in the communication channels if garbage collection (Discard Session in Fig. 11) also sent a SESSION_CLOSED message, so that was removed.

The model presented in the previous section is the final model which has been analyzed and verified to be without errors. The presented model has 2 dead states. In them, all clients are in the Done state, all providers are Shut Down and both AllProviders and AllSessions contain empty lists. Available Sessions contains either 0 or 1, corresponding to the number of unused session identifiers in the model (each client has to consume at least one session). All other places are empty. These dead states therefore exactly exhibit the desired behavior: no

clients or providers are stuck, all sessions are removed, and all channels are empty. Analysis has fixed three problems in the model; two of these would most likely also have shown themselves in an implementation and one made an implicit assumption explicit in the model. This is of course only a single instance of the model, but we have also verified it for other configurations, and as the model has no limits pertaining to the configuration, we are confident the protocol works in all configurations.

5   New Implementation

We have implemented the model described in Sect. 3 and analyzed in Sect. 4 as a service in the process mining tool ProM [9]. The entire implementation was done by one person in two days.

A provider has to implement the interface in Figure 21. The interface shown is slightly simplified as we provide a little more meta-information for session setup and queries in the implementation, but this is not important for our discussion. The accept method is invoked whenever a client creates a session. The session corresponds to a session in the model, and the queryLanguages parameter describes which query languages will be used (not modeled). The accept method returns a boolean indicating whether the provider is willing to handle the session (corresponds to the STARTED/SESSION_ENDED messages). destroy is called when a session is shut down. The simple, comparison, predict, and recommend methods implement the four kinds of queries. They are parametrized with a result type, R, and a query language type L (a query is not necessarily a string, but can also be structured), and each method takes a session and a query as parameters as well as a boolean indicating whether the execution is done (can have impact on the result of queries; for example a Declare process may have temporarily violated constraints, which is allowed unless the execution has terminated). Providers may be called with different session parameters from different threads, so providers should not store data locally between calls, but instead store all information in the Session object.

The OSS has methods for adding and removing providers, addProvider and removeProvider, which each take a Provider as parameter. A client class exists with an interface similar to the Provider in Fig. 21.

```
1  public interface Provider extends Serializable {
2      boolean accept(Session session, List<String> queryLanguages);
3      void destroy(Session session);
4
5      <R, L> R simple(Session session, L query, boolean done);
6      <R, L> Prediction<R> comparison(Session session, L query, boolean done);
7      <R, L> Prediction<R> predict(Session session, L query, boolean done);
8      <R, L> Recommendation<R> recommend(Session session, L query, boolean done);
9  }
```

Fig. 21. Provider Interface.

Monitoring Declare Models. In the new OSS a client instance and a provider are paired only if, inspecting the meta-data available in the session request, the provider chooses to setup the session. This ensures a client is only bound to providers able to provide meaningful results. Meta-data sent on setting up a session allows clients to customize providers by setting configuration parameters. For instance, in the Declare Monitor, each client can set the Declare model which should be used for monitoring, allowing clients executing workflows from different models to use the same server.

Using sessions, a provider can store case-specific information, such as the current state of each Declare constraint. Starting from the current recorded state, the Declare Monitor is able to compute the new state for each constraint when events occur without replaying the entire partial trace each time. This is crucial for the implementation of a run-time monitor which is no longer reduced to an expensive statical checker.

In the new implementation of the OSS, a client can inform the provider that a trace is completed. The Declare Monitor can use this information to raise a violation for all those constraints which are in a pending state when the data stream completes.

6 Conclusion and Future Work

We have presented a new protocol for operational support within business process management supporting sessions, thereby alleviating problems found during development of a real-life provider. The protocol is the result of iterative prototyping and state-space analysis using coloured Petri nets. In addition to the developed protocol, we believe the techniques developed during the prototyping are generally applicable. This includes a very compact state representation for general CPN models, allowing explicit analysis of state-spaces with more than 10^8 states, and demonstrating that model alterations and use of priorities to reduce concurrency reduce the state-space sufficiently. We believe that by building a prototype using a formal model instead of a textual specification or a direct implementation has led to a much clearer and better protocol as well as much faster development. Analysis revealed two major and one minor problems in the protocol which were fixed before implementation. Using the formal model as blueprint for the implementation has made the implementation next to trivial, as evidenced by the implementation time frame of two person days.

Future work includes extending the protocol with a cross-session cache, which can, e.g., be used by the Declare Monitor to store the representation of a Declare Model used for monitoring (which is expensive to create). We do not expect this to have a major impact on the protocol. Furthermore, we have only considered a single-client/single-case scenario here, where a single client is working on a single case. It would also be interesting to consider the cases where two or more clients work on a single case, where a single client works on multiple cases, and where multiple clients work on multiple (not necessarily the same) cases. This should be possible by allowing sharing of sessions, which again would require

authentication, and by extending the protocol with a means for a provider to easily consider multiple sessions at once.

It would be interesting to implement the current compact state representation in ASAP [11] to evaluate the performance of the representation without the overhead of communication between two processes. Also, while we can reduce the overhead per state a bit more, we do not believe that we can explicitly represent states much smaller than now. It would be interesting to instead use a symbolic representation (even if states are calculated explicitly), e.g., using BDDs [1].

Acknowledgment: The authors wish to thank Marco Montali for his input to the design of the new operational support protocol.

References

1. R.E. Bryant. Graph Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
2. S. Christensen and L.M. Kristensen. State Space Analysis of Hierarchical Coloured Petri Nets. *Petri Net Approaches for Modelling and Validation*, pages 1–16, 2003.
3. CPN Tools webpage. Online: `cpntools.org`.
4. S. Evangelista, M.Westergaard, and L.M. Kristensen. The ComBack Method Revisited: Caching Strategies and Extension with Delayed Duplicate Detection. *ToP-NoC*, 3:189–215, 2009.
5. G.J. Holzmann. An Analysis of Bitstate Hashing. *FMSD*, 13:289–307, 1998.
6. K. Jensen and L.M. Kristensen. *Coloured Petri Nets – Modelling and Validation of Concurrent Systems*. Springer, 2009.
7. T. Mailund and M. Westergaard. Obtaining Memory-Efficient Reachability Graph Representations Using the Sweep-Line Method. In *Proc. of TACAS'04*, volume 2988 of *LNCS*, pages 177–191. Springer, 2004.
8. M. Pesic, H. Schonenberg, and W. M. P. van der Aalst. DECLARE: Full Support for Loosely-Structured Processes. In *Proc. of EDOC'07*, page 287, 2007.
9. Process mining webpage. Online: `processmining.org`.
10. A. Rozinat, M. T. Wynn, W. M. P. van der Aalst, A. H. M. ter Hofstede, and C. J. Fidge. Workflow Simulation for Operational Decision Support. *Data Knowl. Eng.*, 68:834–850, 2009.
11. M. Westergaard, S. Evangelista, and L.M. Kristensen. ASAP: An Extensible Platform for State Space Analysis. In *Proc. of ATPN'09*, volume 5606 of *LNCS*. Springer, 2009.
12. M. Westergaard and L.M. Kristensen. The Access/CPN Framework: A Tool for Interacting With the CPN Tools Simulator. In *Proc. of ATPN'09*, volume 5606 of *LNCS*. Springer, 2009.
13. P. Wolper and D. Leroy. Reliable Hashing without Collision Detection. In *Proc. of CAV'93*, volume 697 of *LNCS*, pages 59–70. Springer, 1993.