WorkflowFM: A Logic-Based Framework for Formal Process Specification and Composition

WorkflowFM: A logic-based framework for formal process specification and composition

Petros Papapanagiotou and Jacques Fleuriot

School of Informatics, University of Edinburgh
10 Crichton Street, Edinburgh EH8 9AB, UK
`ppapapan@inf.ed.ac.uk`, `jdf@inf.ed.ac.uk`

Abstract. We present a logic-based system for process specification and composition named WorkflowFM. It relies on an embedding of Classical Linear Logic and the so-called proofs-as-processes paradigm within the proof assistant HOL Light. This enables the specification of abstract processes as logical sequents and their composition via formal proof. The result is systematically translated to an executable workflow with formally verified consistency, rigorous resource accounting, and deadlock freedom. The 3-tiered server/client architecture of WorkflowFM allows multiple concurrent users to interact with the system through a purely diagrammatic interface, while the proof is performed automatically on the server.

Keywords: process modelling, workflows, theorem proving, classical linear logic

1 Introduction

Flowcharts, UML [19], and BPMN [16] are among the most commonly used languages to describe process workflows. These focus more on providing an expressive and intuitive representation for the user as opposed to an executable and verifiable output. In contrast, process calculi such as the π-calculus [15] have fully formal semantics, allowing an array of possibilities for further analysis and reasoning of the constructed models, but making it harder for non-expert users to model complex real-world systems effectively.

In this work, we introduce WorkflowFM as a tool for formal process modelling and composition. In this, processes are specified using Classical Linear Logic (CLL) [9], composed via formal proof, and then rigorously translated to executable process calculus terms. This is performed within the verified environment of the proof assistant HOL Light [11], thus enabling the development of correct-by-construction workflow models. In addition, WorkflowFM employs a client-server architecture, which enables remote access, and incorporates a fully diagrammatic user interface at the client side. These allow users to harness the reasoning power of WorkflowFM and its systematic approach to formal process modelling in an inuitive way and without the need for expertise in logic or theorem proving. WorkflowFM is actively being used to model healthcare processes and patient pathways in collaboration with a number of clinical teams [17, 18, 2].

2 Logic-based process modelling

We begin the description of WorkflowFM by breaking it down to its theoretical foundations, including the proofs-as-processes paradigm, CLL as a process specification language, logical inference for rigorous process composition, the automation required to facilitate workflow development, and the employed diagrammatic visualisation.

2.1 The proofs-as-processes paradigm

The proofs-as-processes paradigm [1, 3] involves a mapping between CLL proofs and process calculus terms, similar to the Curry-Howard correspondence [12] between intuitionistic logic proofs and the λ-calculus. Bellin and Scott analyse this mapping by giving a corresponding π-calculus term for the conclusion of each of the CLL inference rules. As the inference rules are used in a CLL proof, a π-calculus term corresponding to that proof is built based on these mappings. At the end of the proof, it is guaranteed that applying the possible reductions in the resulting π-calculus term corresponds to the process of cut-elimination in the proof. This means that the cut-free version of the proof corresponds to an equivalent π-calculus term that cannot be reduced further. As a result, since π-calculus reductions correspond to process communications, any π-calculus process that corresponds to a CLL proof is inherently free of livelocks and deadlocks.

2.2 Process specification using linear logic

In Linear Logic, as proposed by Girard [9], the emphasis is not only on the truth of a statement, but also on formulas that represent resources. For example, in order to achieve a proof, formulas cannot be copied or deleted (no weakening or contraction), but rather must be *consumed* as resources.

The current work uses the multiplicative additive fragment of propositional CLL without units (MALL), which allows enough expressiveness for simple processes while keeping the reasoning complexity at a manageable level. More specifically, we use a one-sided sequent calculus version of MALL [21]. This simplifies the process specifications and reduces the number of inference rules, which can make automated proof search more efficient.

In this particular version of MALL, linear negation $(\cdot^\perp)$ is defined as a syntactic operator, so that, for example, both A and $A^\perp$ are considered atomic formulas. In addition, de Morgan style equations (e.g. $(A \otimes B)^\perp \equiv A^\perp \,\invamp\, B^\perp$ and $(A \oplus B)^\perp \equiv A^\perp \,\&\, B^\perp$) are defined for linear negation (rather than inference rules). These equivalence equations demonstrate a symmetry in CLL where each operator has a dual. This duality can be exploited in order to represent the information flow of the corresponding resources [3]. We choose to treat negated literals, multiplicative disjunction $(\invamp)$, and additive conjunction $(\&)$ as **inputs**, and positive literals, multiplicative conjunction $(\otimes)$, and additive disjunction $(\oplus)$ as **outputs**.

Based on this distinction, the semantics of the CLL operators can be given an intuitive interpretation where propositions correspond to resources:

- Multiplicative conjunction or the *tensor* operator $(A \otimes B)$ indicates a simultaneous or parallel output of A and B.
- Additive disjunction or the *plus* operator $(A \oplus B)$ indicates that either of A or B are produced but not both. When representing processes, additive disjunction can be used to indicate alternative outputs, including exceptions.
- Multiplicative disjunction or the *par* operator $(A \,\invamp\, B)$ is the dual of multiplicative conjunction and represents simultaneous or parallel input.
- Additive conjunction or the *with* operator $(A \,\&\, B)$, can similarly be interpreted as the dual of additive disjunction, i.e. the representation of optional input.

As an example, based on this interpretation, a process with inputs A and B and outputs C and D can be specified by the CLL sequent $\vdash A^\perp,\; B^\perp,\; C \otimes D$.

In order for CLL sequents to represent meaningful specifications of processes, we impose some restrictions in their form. More specifically, each formula in the sequent can only consist of inputs

(negative literals, $\invamp$, and &) or outputs (positive literals, $\otimes$, and $\oplus$). Moreover, we only allow a single (potentially composite) output formula[1]. We also remark that each CLL specification has a corresponding process-calculus specification based on the proofs-as-processes paradigm.

2.3 Composition via proof

So far, we have described how CLL sequents (under some well-formedness limitations) correspond to specifications of processes. Naturally, the CLL inference rules can be applied to sequents in ways that correspond to rigorous transformation and composition of processes.

The inference rules of our selected version of CLL are presented in Fig. 1. Each rule has an

$$\frac{}{\vdash A,\ A^{\perp}}\ Id \qquad\qquad \frac{\vdash \Gamma,\ C \quad \vdash \Delta,\ C^{\perp}}{\vdash \Gamma,\ \Delta}\ Cut$$

$$\frac{\vdash \Gamma,\ A \quad \vdash \Delta,\ B}{\vdash \Gamma,\ \Delta,\ A \otimes B}\ \otimes \qquad\qquad \frac{\vdash \Gamma,\ A,\ B}{\vdash \Gamma,\ A \invamp B}\ \invamp$$

$$\frac{\vdash \Gamma,\ A}{\vdash \Gamma,\ A \oplus B}\ \oplus_L \qquad \frac{\vdash \Gamma,\ B}{\vdash \Gamma,\ A \oplus B}\ \oplus_R \qquad \frac{\vdash \Gamma,\ A \quad \vdash \Gamma,\ B}{\vdash \Gamma,\ A \,\&\, B}\ \&$$

Fig. 1: The one-sided sequent calculus versions of the CLL inference rules.

intuitive correspondence to a process transformation (unary rules) or composition (binary rules) that we describe next.

- **Id** (Identity): This axiom introduces a process that receives some resource and immediately sends it forward. Such a process is known as an *axiom-buffer*.
- $\otimes$: This composes two processes *in parallel* (to yield a single composite, parallel output).
- $\invamp$: This rule combines two inputs of a process into a single, parallel input.
- $\oplus$: This transforms the output of a process by adding an alternative output (which is never produced in practice).
- &: This rule composes two processes *conditionally*. One input from each process becomes an option in a new, composite input. Either of the two original processes will be executed depending on which option is provided at runtime.
- **Cut**: The *Cut* rule composes two processes *sequentially*. The output of one process is connected to the matching input of another.

We can create complex process compositions in a rigorous way by applying combinations of these primitive rules. The properties of CLL and the proofs-as-processes paradigm provide the following key guarantees of correctness for any composite process we construct in this way:

1. Resources are consistently matched together. Processes that do not match are never connected.
2. Resources are always accounted for systematically, since no contraction or weakening is allowed.
3. The underlying *process-calculus* specification corresponding to the construction is free of dead-locks and livelocks.

---

[1] The subtle reason for this restriction is that the cut rule (corresponding to a sequential composition of processes) allows only a single formula to be cut (connected) between 2 processes.

2.4 Automation

Constructing meaningful process compositions usually requires the application of a large number of primitive steps using the CLL inference rules. This can be impractical, especially for a system that aspires to have users who may not be familiar with CLL or theorem proving more generally. For this reason, we developed 3 high-level automated procedures that perform the necessary proof steps to achieve basic compositions between 2 processes in an intuitive way. We call these procedures composition *actions*, and they are introduced as follows:

- The `TENSOR` action performs a *parallel* composition. Given 2 processes, it uses the $\otimes$ rule to compose them so that they are executed and provide their outputs in parallel.
- The `WITH` action performs a *conditional* composition. Given 2 processes `P` and `Q`, it requires a selected input $A^\perp$ from `P` and a selected input $B^\perp$ of `Q`. It then makes use of the & rule to construct a composite process with input $(A \oplus B)^\perp$. In this, if A is provided at runtime then `P` is executed, otherwise if B is provided then `Q` is executed.
- The `JOIN` action composes 2 given processes `P` and `Q` *sequentially*. It requires a selected subterm of the output of `P` and a *matching* subterm of an input of `Q`. For example, if `P` has output $A \otimes B$ and `Q` has an input $(C \oplus A)^\perp$ then the user can select the left branch A of the output and the matching right branch A of the input. It then makes use of the *Cut* rule to construct a composite process where `P` is executed first, its output A is fed to `Q` and then `Q` is executed.

These implemented actions go well-beyond the limited capabilities of their respective main inference rules ($\otimes$ rule, & rule, and *Cut* rule). In non-trivial cases, a number of inference rules is applied to first transform the processes into a form that is appropriate for use with the main composition rule. It is often the case, particularly in the `JOIN` action, that resources that cannot be handled directly by the involved processes (e.g. by the receiving process in a sequential composition) are automatically *buffered* with the use of axiom-buffers (introduced in Section 2.3).

In addition, we have implemented and integrated an automated CLL prover, based on proof strategies developed by Tammet [20], so that the system can handle cases where complex formulae need to be rearranged in particular ways. For example, the formula $A \otimes B$ can be rearranged to $B \otimes A$ by automatically proving the lemma $\vdash (A \otimes B)^\perp, \; B \otimes A$. Note that, from the process specification perspective, $A \otimes B$ and $B \otimes A$ have the equivalent intuitive interpretation of 2 parallel outputs A and B. However, at the rigorous level of the embedded logic, this reasoning needs to be performed explicitly (yet remain hidden from the user).

The resulting composite processes from each of the 3 actions have their own CLL and process calculus (corresponding to the CLL proof) specifications. These can be used as building blocks for further complex, action-based compositions.

2.5 Visualisation

Although the CLL and process specifications of a constructed composite process describe its structure in a complete and verified way, they may be difficult to grasp by the uninitiated. Their syntax can be difficult to follow, especially for large compositions, even for experts in logic and process algebras.

For this reason, we developed a visual representation and construction that aims to capture the intuitive interpretation of the logical specification and inference steps. The user can then develop

correct-by-construction workflows by interacting with the system visually (see Section 4.3) without the need for expertise in CLL or theorem proving.

The representation involves a simple left-to-right, labelled box notation to represent processes. Dangling edges stand for the inputs and outputs of a process, with solid lines representing parallel resources (i.e. types connected by $\otimes$, $\invamp$, or separate inputs) while dashed lines represent optional resources (i.e. types connected by $\oplus$ or $\&$) as shown in the examples of Fig. 2.

(a) $\vdash X^{\perp},\ Y^{\perp},\ A \otimes B$

(b) $\vdash (X \oplus Y)^{\perp},\ Z^{\perp},\ A \oplus B$

(c) $\vdash X^{\perp},\ A \otimes (B \oplus C)$
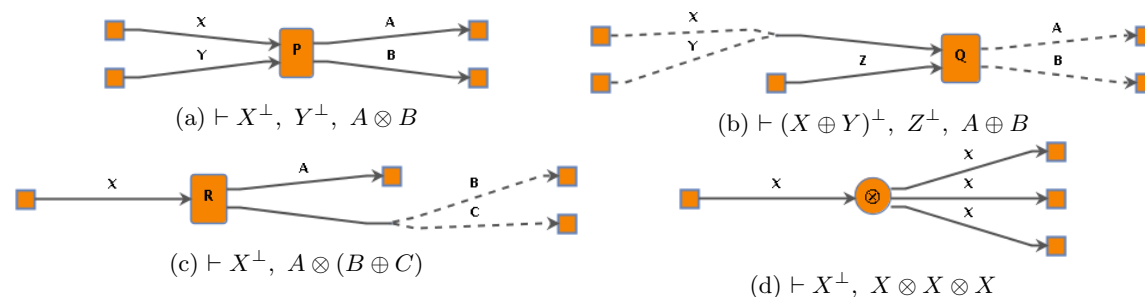
(d) $\vdash X^{\perp},\ X \otimes X \otimes X$

Fig. 2: Examples of specifications of atomic processes in CLL and their corresponding visualisations.

It is worth noting the special round node used for the representation of the so-called *copy nodes* (see Fig. 2d). These are processes that explicitly replicate a resource (assuming this is possible) since this is not allowed to happen within CLL but may be desirable in some scenarios (e.g. creating a copy of an electronic document).

As processes are composed by clicking on edges and boxes to form complex workflows, we adopt a couple of additional visual constructs. Firstly, some resources that are not explicitly handled by a particular process, but rather buffered through an axiom-buffer, are represented using grey edges. Secondly, the output of a composition, as constructed through rigorous CLL reasoning, may be a complex combination of the resources (inputs and outputs) of the component processes. In order to represent these combinations visually, we use triangular nodes that are similar to the decision/merge ones of UML and the gateways of BPMN (see Fig. 3 of Section 3 for an example).

Mapping logical composition steps to such a diagrammatic representation is non-trivial because of the complexity and size of the generated proofs and the particularities of CLL. In order to accurately represent the constructed workflow, the logical engine records important meta-data during each proof step (see Section 4.4 for more details).

3 Example

In order to demonstrate the capabilities of our system and the challenges faced, let us consider a hypothetical example from the healthcare domain. Assume a process `DeliverDrug` with 3 inputs: *Patient* (a resource containing the patient information), *Dosage* (a resource containing information about the drug dosage), and *NurseTime* (a number of allocated hours for a nurse to deliver the drug). Also assume this process can have 2 alternative outcomes: *Treated* in the case where the treatment was successful **or** *Failed* in the case where the drug delivery had no effect. Let us also introduce the process `Reassess`, during which a clinician reassesses the patient if the drug delivery

failed. Its inputs are *Failed* and *ClinTime* (corresponding to the reserved clinical time for this task) and the output is *Reassessed*. In CLL, the 2 processes have the following specifications:

$$\vdash Patient^{\perp}, \; Dosage^{\perp}, \; NurseTime^{\perp}, \; Treated \oplus Failed$$

$$\vdash Failed^{\perp}, \; ClinTime^{\perp}, \; Reassessed$$

We now want to compose the 2 processes into a composite one where the drug failure is handled by `Reassess`. We can use the `JOIN` action, selecting the optional output *Failed* of `DeliverDrug` and the matching input $Failed^{\perp}$ of `Reassess`. The reasoner will automatically connect the 2 processes in sequence by generating the following proof:

$$
\cfrac{
\cfrac{
\cfrac{
\cfrac{\overline{\vdash Treated^{\perp}, \; Treated} \; Id \quad \overline{\vdash ClinTime^{\perp}, \; ClinTime} \; Id}{\vdash Treated^{\perp}, \; ClinTime^{\perp}, \; Treated \otimes ClinTime} \; \otimes
}{\vdash Treated^{\perp}, \; ClinTime^{\perp}, \; (Treated \otimes ClinTime) \oplus Reassessed} \; \oplus_L
}{}
}{}
\tag{1}
$$

$$
\cfrac{
(1) \quad \cfrac{\cfrac{\overline{\vdash Failed^{\perp}, \; ClinTime^{\perp}, \; Reassessed} \; Reassess}{\vdash Failed^{\perp}, \; ClinTime^{\perp}, \; (Treated \otimes ClinTime) \oplus Reassessed} \; \oplus_R}{}
}{\vdash (Treated \oplus Failed)^{\perp}, \; ClinTime^{\perp}, \; (Treated \otimes ClinTime) \oplus Reassessed} \; \&
\tag{2}
$$

$$
\cfrac{
\cfrac{\overline{\vdash Patient^{\perp}, \; Dosage^{\perp}, \; NurseTime^{\perp}, \; Treated \oplus Failed} \; DeliverDrug \quad (2)}{}
}{\vdash Patient^{\perp}, \; Dosage^{\perp}, \; NurseTime^{\perp}, \; ClinTime^{\perp}, \; (Treated \otimes ClinTime) \oplus Reassessed} \; Cut
\tag{3}
$$

The result of this composition proof demonstrates the systematic resource tracking accomplished through the CLL rules. A human relying on the textual description of the processes might intuitively think that the composition of `DeliverDrug` and `Reassess` can have 2 possible outcomes: either the patient was treated by the drug (*Treated*) or they were reassessed (*Reassessed*). However, the outcome of the formal composition tells a slightly different story. The output $(Treated \otimes ClinTime) \oplus Reassessed$ indicates that, in the case where the patient was treated (*Treated*) there was also some unused clinical time (*ClinTime*) left over (which would have been used for reassessment in the case of failure).

Systematically accounting for such unused resources is non-trivial, especially considering larger workflows with tens or hundreds of processes and many different outcomes. The CLL inference rules enforce this by default and the proof reflects the level of reasoning required to achieve this. Part (1) in the proof is essentially dedicated to constructing a buffer for *Treated* and *ClinTime* as they remain unused in one of the 2 optional cases. The reasoner is capable of constructing such buffers automatically and infer which resources can be used directly or need to be buffered.

The effort to visualise such compositions adds another layer of complexity to the system. The visualisation of this particular example is shown in Fig. 3. In this case, the resources found in the output $(Treated \otimes ClinTime) \oplus Reassessed$ come from 2 different sources: *Treated* is an output of `DeliverDrug`, whereas *ClinTime* and *Reassessed* are associated with `Reassess`. These resources are combined in the final output in a seemingly arbitrary way. Therefore, we are unable to directly connect each output edge to its original source, which is usually the case in other workflow languages. This is the reason behind the introduction of the triangle node as a "terminator" node that performs this combination of resources.
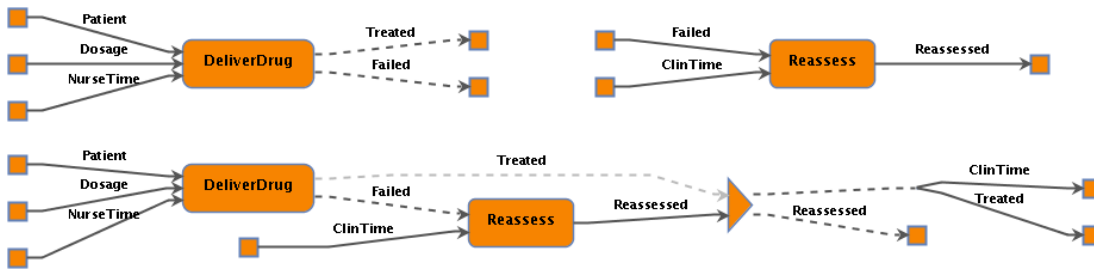
Fig. 3: The visualisation of the `DeliverDrug` and `Reassess` processes (top) and their sequential composition using `JOIN` (bottom).

In addition, $ClinTime$ is *not* an output of `Reassess`, but an unused input that is buffered in one of the two cases. This makes it hard to track where $ClinTime$ came from in the final output. One would be inclined to draw a grey $ClinTime$ edge from `Reassess` to the triangle node, but this might give an erroneous impression that $ClinTime$ is an output of `Reassess` (which could be the case in a different situation). For this reason, we have chosen not to display such an edge.

Our design decisions about the visualisation are still evolving as we carry out more case-studies and endeavour to make the interface as intuitive and straightforward as possible without compromising the correspondence with the logical underpinnings.

4 Architecture

WorkflowFM adopts a server-client architecture consisting of 3 tiers: the reasoner, the server, and the client. Each of the 3 components can be deployed separately and can be connected to each other remotely. More specifically, the reasoner connects to the server through a (local or remote) data pipe (e.g. via an SSH tunnel). A client can then connect to the server via a raw socket. The server effectively screens and relays JSON commands from the client to the reasoner and returns the response from the reasoner in the same way. Moreover, the server may connect with multiple reasoners and distribute the commands between them. Multiple clients can also connect to the server and issue commands concurrently.

This architecture has several advantages over what would otherwise be a single user, stand-alone system, such as the following:

- It allows access through a lightweight, platform independent Java client. HOL Light has multiple system dependencies and can be difficult to install, setup, and run for a non-expert user. In contrast, WorkflowFM's HOL Light based reasoner can be installed and maintained by experts and can remain live and always online.
- Allowing multiple concurrent users on the server makes WorkflowFM more scalable and accessible by a larger array of users.
- Some reasoning tasks (especially for large or complex compositions) may take considerable time (up to a few minutes). Allowing multiple connected reasoners helps improve efficiency and responsiveness, especially during concurrent operations by multiple users.

The overall system architecture coupled with the stacked architecture of the reasoner and the client are visualised in Fig. 4. We explain the structure and functionality of each component next.
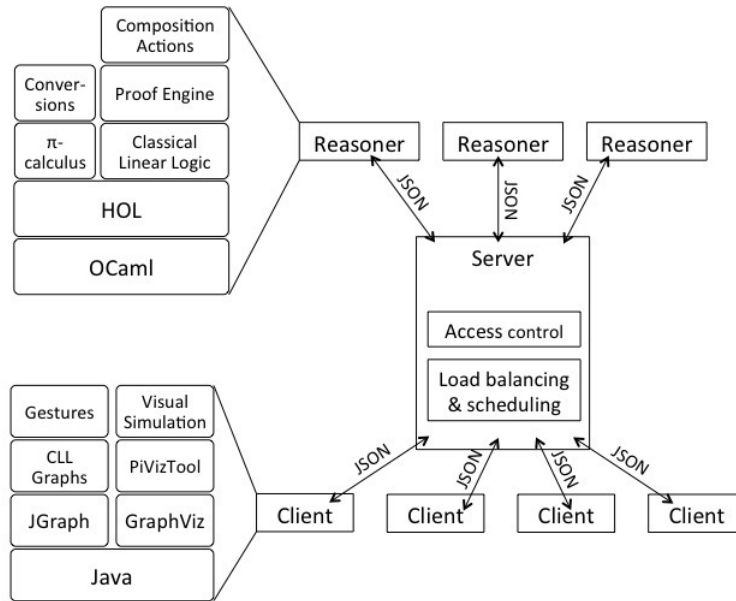
Fig. 4: The 3-tiered server-client architecture of WorkflowFM.

4.1 The reasoner

The reasoner is implemented on top of the interactive proof assistant HOL Light [11], which operates at the OCaml toplevel and provides a flexible, trustworthy environment that conveniently marries theorem proving and programming. Note that, although theorem proving systems for linear logic already exist (e.g. Forum [14] and LINK [10]), we aimed to implement WorkflowFM in a fully formal, trustworthy environment, with an expressive enough meta-logic to support the proofs-as-processes paradigm. HOL Light is a prime example of such an environment.

More specifically we have deeply embedded [5] the following within the Higher Order Logic setting of HOL Light:

1. The syntax (including primitive name binding and substitution) of the π-calculus.
2. The syntax of CLL formulae.
3. The CLL inference rules including their process calculus correspondences.

Given these embeddings, we developed the necessary proof infrastructure to apply the CLL inference rules for both forward and backward inference and the automated actions mentioned in Section 2.4.

The reasoner can be used at two levels. On the one hand, it is integrated in the HOL Light proof engine so that each composition action can be applied as a proof tactic. On the other hand, it has its own independent interface that relies on JSON data structures (see Section 4.4).

4.2 The server

The server, currently written in Java, acts as an intermediate level between the reasoner and the client. Its primary function is to enable the management of multiple clients and multiple reasoners.

Firstly, it is able to connect to multiple instances of the reasoner, both locally and remotely. Thus, in cases where multiple users submit multiple commands at the same time, it can distribute the load among all connected reasoners. This is facilitated by the fact that the reasoner is stateless. Moreover, it caches the results returned by the connected reasoners for each issued command.

Secondly, it manages multiple connections from clients. It exposes an open public socket that can receive connections and manages them, e.g. by mapping reasoner responses to the appropriate connection, handling timeouts and reconnections, and logging traffic and miscellaneous statistics.

4.3 The client

The client is a Java-based graphical user interface, developed using the JGraph library [13], that renders the process specifications in the diagrammatic way described in Section 2.5.

Additionally, the client allows the construction of such process specifications and compositions via interactive gestures. Atomic process specifications are constructed in a visual way, starting from a process with one input and one output. The user augments this basic process specification e.g. by adding new resources, creating new types of branches (start an $\oplus$ branch in a $\otimes$ node and vice-versa), renaming resources and so on, via mouse-driven clicks.

Once the user is happy with the visual representation of the process they wish to construct, the specification is sent to the reasoner for verification. All such validated specifications (which include both the CLL and the process-calculus components) are stored in a list in the interface. From there they can be picked and composed in the current visual *workspace* using mouse gestures that implement the `TENSOR`, `WITH` and `JOIN` actions.

The composition results returned by the reasoner are stored as intermediate processes in their corresponding workspace. We note here that the WorkflowFM interface provides a multi-tabbed workspace environment that enables the user to develop different composition scenarios simultaneously without their respective intermediate steps interfering. Also note that in order to draw accurate and unambiguous representations of the composed workflow, the client expects some meta-data from the reasoner, as explained in more detail in Section 4.4.

Overall, the user-interface is driven by mouse gestures performed by the user and captured by handlers based on the principles of event-driven programming [8]. Through these, each gesture initiates a proof task in the reasoner resulting in a verified response. This can be thought of as *event-driven theorem proving*.

As previously mentioned, the reasoner itself is stateless and therefore the client is fully responsible for storing its state. More specifically, the client stores the atomic processes, stored composite processes and the composition actions required to construct them, and the workspaces with the intermediate compositions performed in each one.

Since the state is stored locally, it can potentially be modified to enable the maintenance of the constructed models. For example, the user can rename processes or resources, modify atomic process specifications, or reload the composition steps for a composite process in order to make alterations. However, such modifications may break the correctness of the corresponding process specification, as well as the specifications of all composite processes that use this process as a component. Thus, the client flags all possibly affected processes and forces the user to *verify* them again through the reasoner in order to maintain the various guarantees of correctness.

Finally, the client provides additional functionality, such as the creation of screenshots and the visual simulation of process calculus terms (as returned by the reasoner) using the PiVizTool [4] and GraphViz [7].

4.4 Data structures

The 3 components of WorkflowFM communicate using JSON messages sharing a common schema. This allows them to be decoupled and thus developed and maintained independently. In a nutshell, the JSON data schema describes the following structures:

- Resources;
- Channels and process-calculus specifications;
- Composition actions with the process names and selected resources as arguments;
- Processes, each described as a record with fields describing (among other things) its name, input resources (paired with its process calculus channel), CLL specification, and so on;
- A composition state that records buffered resources, input and output provenance entries, etc. to ensure the appropriate visualisation of the connections between the processes;
- Commands that were issued to the reasoner along with their arguments;
- Responses from the reasoner to be interpreted by the client.

It is worth noting that the JSON data structures already provide a significant level of abstraction from the mechanics of the theorem proving backend. Our aim was to construct an API that incorporates all the necessary data (for both the reasoner and the client), but requires limited to no experience with logic or theorem proving.

5 Conclusion

In summary, WorkflowFM is a logic-based framework for workflow modelling. It employs an event-driven theorem proving approach where mouse gestures on graphs in the GUI trigger custom-built, automated proof procedures. This aims to alleviate much of the complexity inherent to interactive theorem proving in this domain. Its architecture consists of 3 distinct components. The reasoner, implemented within the proof assistant HOL Light, relies on the proofs-as-processes paradigm to allow formal process specification in CLL and correct-by-construction process composition through logical inference. The Java client visualises the CLL workflows and enables an intuitive user interaction. The server acts as a relay between multiple reasoners and multiple clients.

It is worth remarking that some of the other salient aspects of the system have not been covered in the current paper due to space limitations. These include the generation of executable code for the composed workflows and the ability to accommodate more advanced process calculus translations of CLL (e.g. with session types [6] or with the process calculus *CP* [22]).

Our plans for future work include efficiency optimisations for the reasoner, improved authentication, user identification, access control, and security for the server, and alternative implementations of the client (such as web or mobile applications).

We believe WorkflowFM is a flexible system that successfully hides the complexity and formality of its theorem proving foundations. In so doing, it makes an inherently complicated process lightweight and approachable by non-expert users. As a result, WorkflowFM is proving to be an effective tool for general purpose process modelling, with guaranteed levels of consistency and resource accounting not currently achievable in other workflow tools.

References

1. Abramsky, S.: Proofs as processes. Theoretical Computer Science 135(1), 5–9 (1994)
2. Alexandru, C., Clutterbuck, D., Papapanagiotou, P., Fleuriot, J., Manataki, A.: A Step Towards the Standardisation of HIV Care Practices (11 2016)
3. Bellin, G., Scott, P.: On the π-calculus and linear logic. TCS 135(1), 11–65 (1994)
4. Bog, A., Puhlmann, F.: A Tool for the Simulation of π-Calculus Systems. Tech. rep., Open.BPM, Geschäftsprozessmanagement mit Open Source-Technologien, Hamburg, Germany (2006)
5. Boulton, R.J., Gordon, A.D., Gordon, M.J.C., Harrison, J., Herbert, J., Tassel, J.V.: Experience with embedding hardware description languages in HOL. In: Stavridou, V., Melham, T.F., Boute, R.T. (eds.) TPCD. IFIP Transactions, vol. A-10, pp. 129–156. North-Holland (1992)
6. Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. LNCS pp. 222–236 (2010)
7. Ellson, J., Gansner, E., Koutsofios, L., North, S.C., Woodhull, G.: Graphviz—Open Source Graph Drawing Tools. Springer Berlin Heidelberg, Berlin, Heidelberg (2002)
8. Ferg, S.: Event-Driven Programming: Introduction, Tutorial, History. `http://eventdrivenpgm.sourceforge.net/` (2016)
9. Girard, J.Y.: Linear logic: its syntax and semantics. In: Girard, J.Y., Lafont, Y., Regnier, L. (eds.) Advances in Linear Logic. No. 222 in London Mathematical Society Lecture Notes Series, Cambridge University Press (1995), `http://iml.univ-mrs.fr/~girard/Synsem.pdf.gz`
10. Habert, L., Notin, J.M., Galmiche, D.: LINK: A proof environment based on proof nets. LNCS pp. 330–334 (2002)
11. Harrison, J.: HOL Light: A tutorial introduction. LNCS pp. 265–269 (1996)
12. Howard, W.A.: The formulas-as-types notion of construction. In: Seldin, J.P., Hindley, J.R. (eds.) To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism, pp. 479–490. Academic Press (1980)
13. JGraph Ltd: The JGraph homepage. `http://www.jgraph.com/` (2013)
14. Miller, D.: Forum: A multiple-conclusion specification logic. TCS 165(1), 201–232 (1996)
15. Milner, R.: Communicating and mobile systems: the π-calculus. Cambridge Univ Presss (1999)
16. Object Management Group: Business Process Model and Notation (BPMN), version 2.0 (2011), `http://www.omg.org/spec/BPMN/2.0/PDF`
17. Papapanagiotou, P., Fleuriot, J.: Formal verification of collaboration patterns in healthcare. Behaviour and Information Technology 33(12), 1278–1293 (10 2014)
18. Papapanagiotou, P., Fleuriot, J.: Modelling and implementation of correct by construction healthcare workflows, pp. 28–39. Lecture Notes in Business Information Processing, Springer (2014)
19. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modelling Language User Guide. Addison-Wesley (1999)
20. Tammet, T.: Proof strategies in linear logic. Journal of Automated Reasoning 12(3), 273–304 (1994)
21. Troelstra, A.S.: Lectures on Linear Logic. CSLI Lecture Notes 29, Stanford (1992)
22. Wadler, P.: Propositions as sessions. In: Proceedings of the 17th ACM SIGPLAN international conference on Functional programming. pp. 273–286. ACM (2012)