Tinker, tailor, solver, proof*

Gudmund Grov
Heriot-Watt University, Edinburgh, UK
G.Grov@hw.ac.uk

Aleks Kissinger
University of Oxford, UK
alek@cs.ox.ac.uk

Yuhui Lin
Heriot-Watt University, Edinburgh, UK
Y.Lin@hw.ac.uk

We introduce Tinker, a tool for designing and evaluating proof strategies based on *proof-strategy graphs*, a formalism introduced by the authors in [7]. We represent proof strategies as open-graphs, which are directed graphs with additional input/output edges. Tactics appear as nodes in a graph, and can be 'piped' together by adding edges between them. Goals are added to the input edges of such a graph, and flow through the graph as the strategy is evaluated. Properties of the edges ensure that only the right 'type' of goals are accepted. In this paper, we detail the Tinker tool and show how it can be integrated with two different theorem provers: Isabelle and ProofPower.

1 Proof strategy graphs – a framework for tinkering

Traditionally, the way to build up a proof from atomic tactics is to chain them together (possibly with some repetition, alternation, etc.), letting later tactics act on sub-goals produced by earlier ones. However, without any extra 'plumbing' to control the flow of goals through a proof, one tends to depend on semantically irrelevant data, such as the *order* in which tactics produce goals to dictate where those goals might end up, which leads to brittleness in proof strategies. For example, consider a case where we expect three sub-goals from tactic t_1, where the first two are sent to t_2 and the last to t_3. A small improvement of t_1 may result in only two sub-goals. This "improvement" causes t_2 to be applied to the second goal when it should have been t_3. The tactic t_2 may then fail or create unexpected new sub-goals that cause some later tactic to fail.

There are a handful of ways one might go about adding this extra plumbing to a proof strategy. The technique provided by the Tinker tool[1], detailed in this paper, makes use of *proof-strategy graphs* (PSGraphs), as defined in [7], which take the notion of goal-plumbing quite literally. Tactics appear as nodes in an open-graph, which is essentially a directed graph with the additional property that we allow dangling edges, i.e. edges without source and/or target nodes. Edges without source or target nodes serve as inputs and outputs to the graph as a whole. One evaluates a PSGraph by placing one or more goal-nodes, each containing a single goal, on input edges of the graph, then applying tactics to consume goals on the in-edges of a tactic-node and producing sub-goals on the out-edges. As a result, the goals appear to flow through the graph, hitting tactics along the way, until they are either consumed (i.e. closed), or reach the output edges of the graph, in which case they become output goals for the overall evaluation. In Figure 1, we show some goals making their way through a PSGraph with two nodes, labelled by the tactics they represent (in this case 'induct' and 'ripple').

Note how a single tactic node can have many out-edges. In principle, applying a tactic could send sub-goals to any of the output edges. However, there will typically only be one particular output that is appropriate for a any particular goal. When plumbing a house, pipes comes in all sizes and shapes, and

*This work has been supported by EPSRC grants: EP/H023852, EP/H024204 and EP/J001058, the John Templeton Foundation, and the Office of Navel Research.

[1] Available at https://github.com/ggrov/psgraph/tree/uitp14.

C. Benzmüller and B. Woltzenlogel Paleo (Eds.):
Eleventh Workshop on User Interfaces for Theorem Provers (UITP 2014)
EPTCS 167, 2014, pp. 23–34, doi:10.4204/EPTCS.167.5

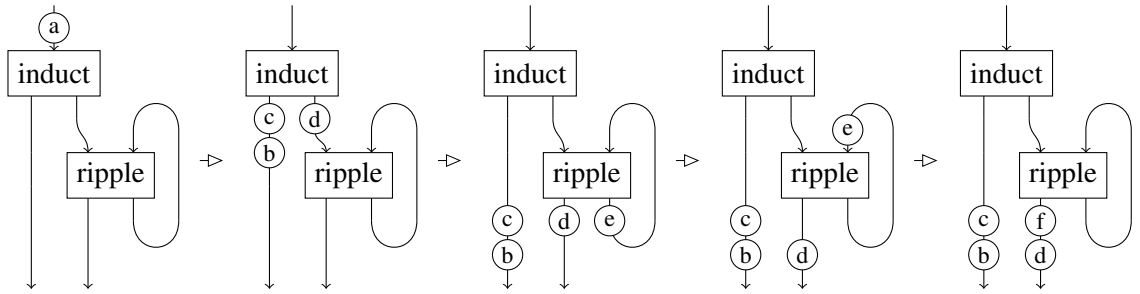
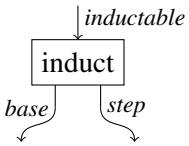


Figure 1: Some goal-nodes (depicted as circles) making their way through a PSGraph

you can only connect the same types of pipes together. The same is true for tactics: they only work for certain goals (although for some tactics this range of goals is rather wide). For example, an 'assumption' tactic expects a hypothesis to be unifiable with the goal, and '$\forall$-intro' expects the goal to start with a $\forall$ quantifier.

To effectively decide where to send goals, we label edges with *goal-types*, which encode certain properties about a goal which dictate how it should then be handled. We then only allow goals to be output on edges with a matching goal-type. For example, the 'induct' tactic might have two output edges: one with type *step*, which matches goals of the form '$?P \implies ?Q$' where $?P$ embeds in $?Q$, and the other with type *base*, which matches everything else. This output types then direct the flow of goals out of the induction tactic in a bigger PSGraph (Figure 2 (left)).



One evaluation step works by applying a single tactic node on a single goal. Here, the goal is consumed from the input edge, the tactic in the tactic node is applied to the goal, and the resulting sub-goals (if any) are sent down the output edges where they match. When all the goal nodes are on output edges of the graph, then it has successfully evaluated. If no output type matches a goal, then evaluation fails. For evaluation this improves robustness of the tactic in two ways: (1) since composition is over the *type of goals*, we avoid the brittleness arising from defining composition in terms of the number of sub-goals or order of sub-goals, and (2) if an unexpected sub-goal arises then evaluation will fail at the actual point of failure as it will not match any of the output types. In general, we allow this evaluation procedure to be non-deterministic by introducing branching whenever a tactic behaves non-deterministically, or a sub-goal produced by a tactic matches more than one output edge. However, with appropriate choice of goal types and evaluation strategy, this branching can be minimised.

Figure 2 (left) highlights an example of a proof strategy employing tactics which rely on specific properties of a goal. For example, *rippling* [2] is a heuristic rewriting technique most commonly used on step cases of inductive proofs, ensuring that each 'ripple' step moves the goal towards the induction hypothesis (IH). This step is repeated until the IH can be applied to simplify or fully discharge the goal – a process called 'fertilisation'. The advantage of rippling is that it is guaranteed to terminate, whilst allowing rewriting behaviour that would not otherwise terminate (e.g. allowing a rewrite rule to be applied in both directions). Termination is ensured by checking that a certain *embedding* property holds for the goal being rippled, while a measure is reduced from a previous goal. Collectively, these properties are captured by a goal type, in this cased called '*can-ripple*'. When a goal is fully 'rippled', then 'fertilisation' is applied.

Proof strategies can easily become very large and complex. In PSGraph, we can reduce this complexity and size by hiding parts of a graph, which is achieved by boxing a sub-graph up into a single vertex. We do this by introducing *graph hierarchies*. A simple example of a hierarchy is shown in Figure 2.

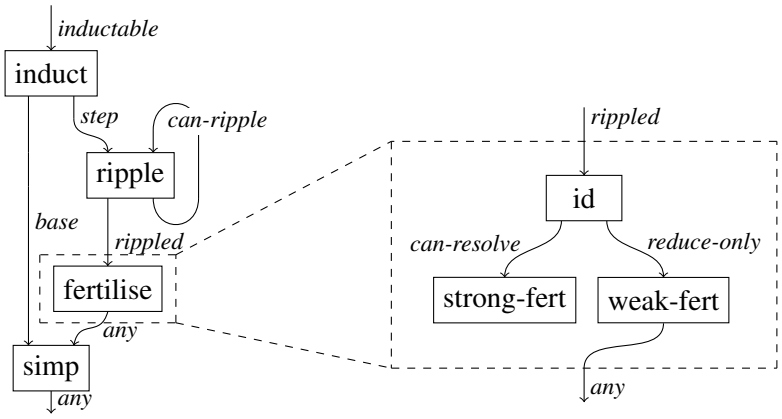


Figure 2: A simple graph hierarchy. The node marked 'fertilise' in this PSGraph is itself a PSGraph, consisting of three atomic tactics.

In this paper we extend [7], by providing details of the architecture and use of the Tinker tool, which implements the PSGraph formalism. In the next section we exhibit the main aspects of the Tinker UI, namely the means by which users can build and evaluate PSGraphs in the theorem provers Isabelle and ProofPower. In section 3, we provide a more detailed overview of the Tinker architecture, including how to integrate Tinker with a new theorem prover.

2 Using Tinker

Currently, Tinker operates in two distinct modes: *evaluation*, where the user employs existing PSGraphs to prove conjectures, and *construction*, where the user builds new PSGraphs. We aim to integrate these two modes in the future, so that strategies can be modified and improved on the fly during evaluation.

2.1 Evaluating PSGraphs

PSGraphs are designed to guide, rather than replace, an existing proof system. Thus they should be seen as a generic, theorem prover-independent tool. This is reflected in Tinker, which currently has interfaces implemented for both Isabelle and ProofPower. Users still use the existing interfaces of those provers, with certain extensions provided by Tinker, notably for building PSGraphs and stepping through graph evaluation (see Figure 8). The latter is primarily used in the course of designing or debugging strategies, so Tinker can also evaluate PSGraphs in non-interactive mode, which behaves like normal tactic evaluation.

2.1.1 Tinkering with Isabelle

Tinker is integrated with Isabelle as a new theory on top of the 'Main' Isabelle/HOL theory[2]. On top of this we have created a new proof method for Isabelle/Isar called psgraph, which can be applied in one of the following ways:

apply (psgraph ⟨graph-name⟩)
apply (psgraph (interactive) ⟨graph-name⟩)

[2]See `https://isabelle.in.tum.de/` for details.

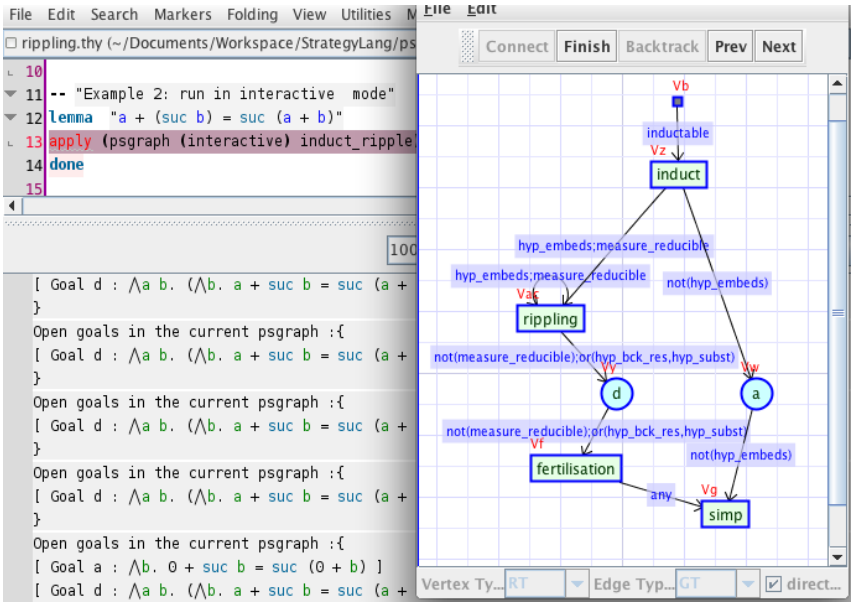


Figure 3: Tinker GUI with Isabelle

apply (psgraph (current))

There are three different modes to work with this tactic. Firstly, if the only argument given is ⟨graph-name⟩, then it enters the 'automatic' mode, which from the user's point of view looks exactly like using any other Isabelle method. The two other modes, 'interactive' and 'current', utilise the Tinker GUI. This enables users to step through the proof and visualise the flow of goal-nodes. Figure 3 shows an example of the 'interactive' mode using the rippling strategy described above. Here, the related goal information will be printed in the Isabelle output panel. The supported actions in the Tinker GUI are 'apply the next box / tactic', 'backtrack', 'replay the previous step' and 'terminate the current evaluation'. If a node represents a nested PSGraph, then a new window is opened showing the nested graph, which the user can evaluate as with the parent graph.

The 'current' mode is used when the user builds a new PSGraph in the Tinker GUI, which is described below. Here, the graph that is currently open in the GUI is used by the psgraph method. The available operations in this mode are the same as those in the 'interactive' mode.

PSGraphs are stored in Isabelle's theory context, so to use a PSGraph ⟨graph-name⟩ in 'interactive' or 'automatic' mode, a graph with the given name first needs to be stored in the theory context.

2.2 Tinkering with ProofPower

The second theorem prover that is supported by Tinker is ProofPower. Here, Tinker has been integrated with ProofPower's subgoal package, used to handle goals and soundness of tactics via the kernel. A PSGraph is executed by the function '**run_psg_goal**', which is invoked in a manner similar to the Isabelle method:

run_psg_goal ⟨goal⟩ ⟨graph-name⟩ auto
run_psg_goal ⟨goal⟩ ⟨graph-name⟩ interactive
run_psg_goal ⟨goal⟩ current

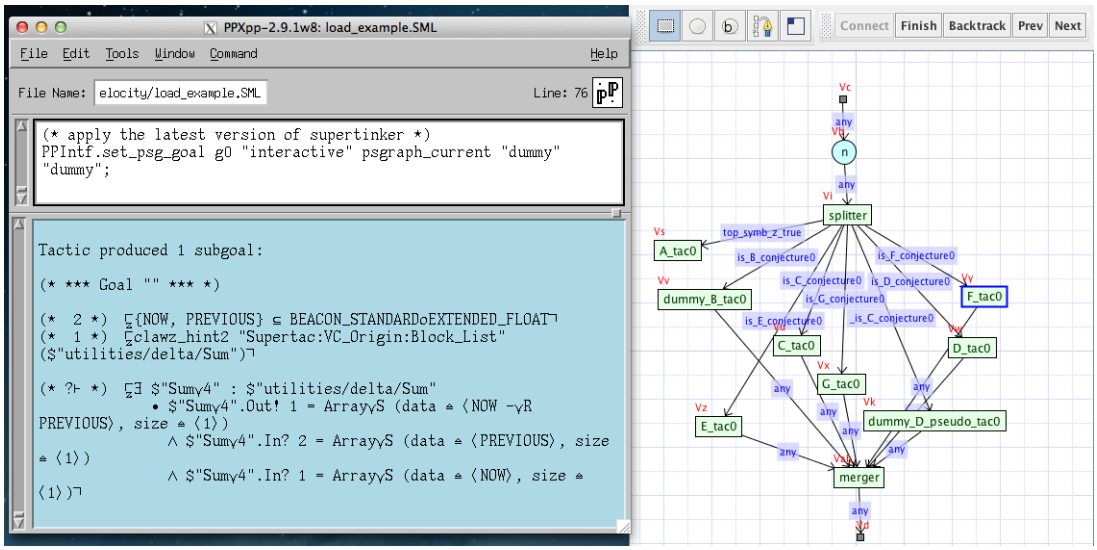


Figure 4: Tinker GUI with ProofPower

Such calls will initiate a proof of ⟨goal⟩ with the PSGraph ⟨graph-name⟩. From this point, Tinker behaves identically to the Isabelle version. Figure 4 shows a screenshot of a PSGraph encoding of SuperTac, a powerful and complex ProofPower tactic consisting of thousands lines of ML code and used by D-RisQ[3] in their ClawZ toolchain [11].

2.3 Building PSGraphs

There are effectively two ways to build a PSGraph to be used by the Tinker system. The first is by drawing graphs using the GUI, and the second is by programming graphs by combining simpler graphs into more complicated ones, via a handful of *graph combinators*.

2.3.1 Tinkering by drawing

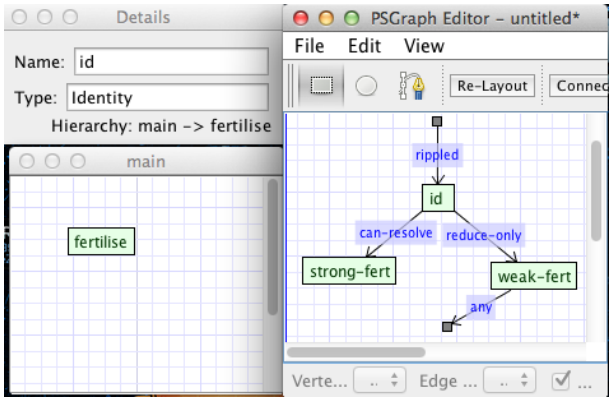


Figure 5: Tinker Drawing GUI

Drawing a PSGraph in the Tinker GUI is straight forward. Using node and edge tools, the user clicks to place tactic boxes and drags lines to connect boxes with edges. Inputs and outputs are represented using 'dummy' vertices (depicted as small grey boxes). Then, double-clicking on nodes or edges allows the user to edit the tactics or goal types, respectively. These tactics and goal types are pre-defined in Tinker, so that they can be looked up by name during evaluation. Also the Tinker GUI allows the user to draw hierarchies as shown in Figure 5. A 'Details' window is available to show the name and type of the selected tactics as well as the path of the current hierarchy.

[3] See `www.drisq.com`.

2.3.2 Tinkering by programming

In addition to being able to draw new PSGraphs, more complex PSGraphs can be built from simpler ones using a handful of graph combinators.

The atomic tactics available to Tinker correspond to tactics provided by the underlying theorem prover, and a key feature of PSGraph is that it provides a **type-safe** method to combine the tactics. Graphs are composed by plugging output edges with input edges. These edges are typed and we can only plug "comparable" types together, where the notion of "comparable" is goal-type specific.

Before an atomic tactic can be included in a PSGraph, it must be equipped with a typing on its inputs and outputs. This is achieved by the function $lift(node\text{-}nm, tac, ins, outs)$, which produces a graph with a single tactic-node named *node-nm*, with a list of input edges *ins* and output edges *outs*. This tactic-node will then be evaluated by calling the underlying tactic function *tac* in the theorem prover.

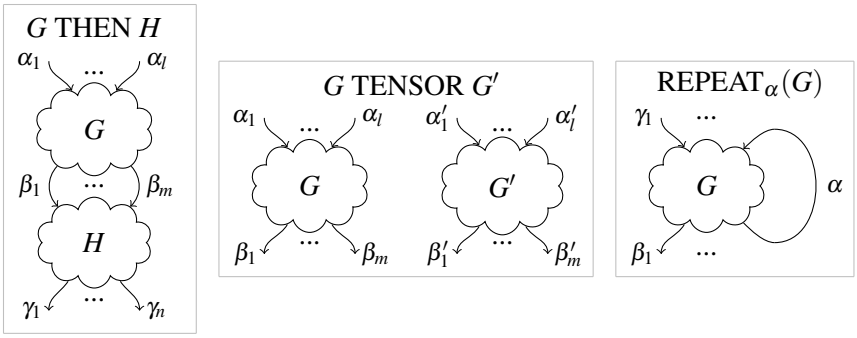


Figure 6: THEN, TENSOR, and REPEAT$_\alpha$ combinators

One of the most common ways to combine tactics is via a *THEN* tactical, where t_1 *THEN* t_2 first applies tactic t_1 then tactic t_2. PSGraphs have a corresponding combinator, also called *THEN*, where *G THEN H* plugs each output edge of G into an input edge of H of the same type. This is illustrated on the left-most diagram of Figure 6. Note that there could be multiple ways of plugging PSGraphs together, so Tinker provides two versions of the combinator: one version that returns a list of all possible ways G could be plugged into H and one version that simply picks one. Further, note that a *maximal plugging* is always done, meaning if two edges can be connected they will be. Any remaining, un-plugged inputs/outputs will become inputs/outputs of the new graph.

Another combinator is *TENSOR*, which puts G and G' side by side in a single graph, without doing any plugging. This is depicted in the middle diagram of Figure 6. The right-most diagram shows how repetition can be naturally represented as a feedback edge. This is can be achieved by invoking the $REPEAT_\alpha$ combinator, which plugs an output of goal type α to an input of the same type. If we think of α as a loop-condition, this provides a graphical analogue to the common *REPEAT_WHILE* or *REPEAT_UNTIL* tacticals. While it is possible to encode the more common *REPEAT* tactical[4] we do not see this as a natural way of working with PSGraphs, as the goal is make a proof strategy developer think about why certain choices have been made, and represent this intuition in the goal types.

Hierarchies are supported in PSGraph, and the *NEST* combinator introduces a hierarchy by "collapsing" the graph into a single graph tactic with the original graph nested. This is illustrated graphically in

[4]To be precise, repeating t until it fails can be encoded using the existing combinators as:

$$REPEAT_\alpha\Big(\big(lift(t,t,[\alpha,\alpha][\alpha])\ TENSOR\ lift(id,id,[],[\beta])\big)\ ORELSE\ \big(lift(id,id,[\alpha,\alpha][\beta])\ TENSOR\ lift(id,id,[],[\alpha])\big)\Big)$$

where *id* is the identity tactic, and α and β are goal types that succeeds for any goal.

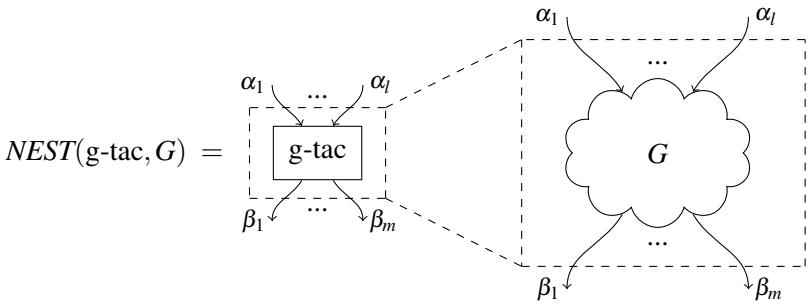


Figure 7: NEST combinator, which collapses a graph G to a single node 'g-tac'

Figure 7. Two generalisations of *NEST* are the *OR* and *ORELSE* combinators, which rather than nesting a single graph, will nest a pair of graphs. When evaluating G *OR* H, Tinker will branch, trying to evaluate both G and H. On the other hand, G *ORELSE* H will only evaluate H if the evaluation of G fails. In the current version of Tinker, G and H must have the exact same number and types of input and output edges, however a direct generalisation is to produce a nested graph node whose inputs are the intersection of the inputs of G and H, and whose outputs are the union of the outputs of G and H.

3 Tinker tool architecture

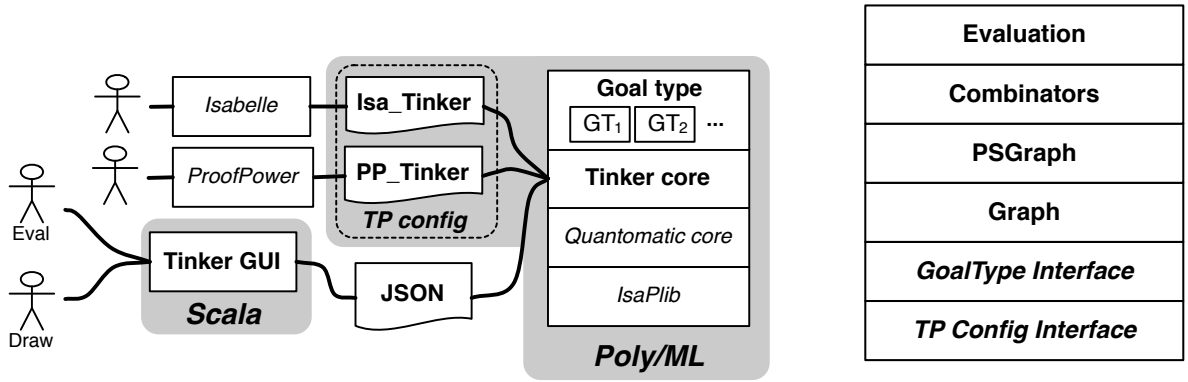


Figure 8: The Tinker architecture (left) and the Tinker core components (right)

Figure 8 (left) shows the architecture of the tool. The tool consists of two overall parts, each are in separate grey boxes. The **core** is where all the "magic" happens, and is implemented in Poly/ML. The **GUI** is implemented in Scala and provides a graphical way of creating, modifying, evaluating and debugging PSGraphs. The Tinker tool is independent of the underlying theorem prover and goal types. In both cases, this extra data is provided via ML structures passed to the Tinker core functors.

Figure 8 (right) shows the main components of the Tinker core, and their dependencies, and each component is detailed below.

3.1 IsaPlib & the Quantomatic core

The Tinker tool started out as a new language to write proof plans in the IsaPlanner proof planner [5] for Isabelle. However, Tinker has evolved considerably since then, and the only parts used are those retained

in an ML library called IsaPlib, which can either co-exist with Isabelle/ML or provide an Isabelle/ML-like environment and adds some extra functionality.

PSGraphs themselves are open-graphs, and evaluation is done by open-graph rewriting. A formalisation and rewrite theory for open-graphs was provided in [6]. The Quantomatic core [9] provides a general purpose open-graph rewriting library on top of IsaPlib, which is employed by Tinker for PSGraph evaluation.

3.2 The Tinker GUI & JSON communication protocol

A JSON protocol is used to communicate between the Scala GUI and Poly/ML core. As seen in Figure 9, the GUI communicates with the core over a socket in a client/server mode. Along with a unique message ID, a JSON message has three relevant fields: *command*, *input* and *output*. The command is a string indicating a command defined by the core. Both input and output are in the JSON format, thus data that needs to be communicated between the core and the GUI must always be serialisable as JSON.

The protocol is implemented on top of the interface provided by Quantomatic, with Tinker specific features to work with graphs (such as evaluation and hierarchies).

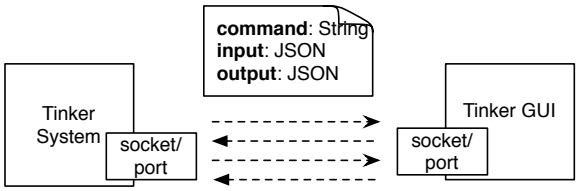


Figure 9: Communication with JSON protocol

3.3 Goal types

There are two perspectives one can take on goal types. The first, which was presented in [7], is to view a goal type simply as a *predicate* on a goal. In this sense, typing provides a mechanism for filtering goals as they are output from tactics to send them to the right place. A goal type also encodes a strategy-writer's expectation for what goals should look like at various points throughout the proof. This can help to make sense of large and complex proof strategies, and more importantly, *why* and *where* an evaluation has failed. This information can then be used to patch the proof strategy, either manually, or automatically via a *proof critic* mechanism (c.f. [2]).

The second perspective on goal types, which subsumes the first, is that they should be used a general mechanism for *guiding* proofs and tactics. Here, instead of just seeing a goal as a predicate it is seen as a (partial) *function* on goal nodes. It is undefined in cases where the type does not match the goal, but when it does match, this function can be used to augment the goal-node with additional information, which can for example highlight relevant lemmas/hypothesis for a tactic. This data can in turn be used later by other tactics. Such goal types can even be initialised automatically via machine-learning techniques. A first approach to this is discussed in [8]. Tinker supports this more general kind of goal type, but for the sake of simplicity, we have chosen to focus mainly on 'predicate-style' types for this paper.

3.3.1 Creating a new goal type – the GoalType interface

To create a new goal type the GoalType signature has to be implemented. Firstly, types

```
type T
type gnode
```

to represent a goal type and a goal node on the graph must be implemented, as well as a matching function:

```
val match :  gnode -> T -> pnode -> gnode option
```

where `match` is the functional view of a goal type, and a predicate will just return the same `gnode` for success and `NONE` if it fails. Note that `gnode` is the goal node on the graph, while `pnode` is an element of the underlying proof representation. The key differences between them is that a `gnode` is on the graph and must be serialisable, while this is not required for a `pnode`. Note that the `gnode`, is the "previous node" (the one that generated the `pnode`) and is used to encode features such as 'measure-reduction' for rippling. We will return to `pnodes` below.

For the 'basic' goal type [7], `T` is a string. The string is assumed to have a set of predicates separated by ';', and for each of these names a matching function has to be created, which is called by `match`. One example is 'top_symbol'. When the substring 'top_symbol' is found then a matching function is called that checks if the top level symbol of the goal matches with a given term. This implementation of goal-types has limited support for logic in the form of 'not' and 'or' connectives. To work with the GUI, a translation between `T` and `gnome` and a JSON representation must also be provided.

3.4 Graph & PSGraph representation

As we mentioned in Section 3.1, Tinker uses the Quantomatic library for graph rewriting. This is done by instantiating a *GRAPHICAL_THEORY* structure called *PSGraph_Theory*. To instantiate a graphical theory, types for graph nodes and edges need to be provided, along with relevant functions for data matching and I/O. Quantomatic then constructs structures for graphs, graph rewrite rules, matching, rewriting, etc.

For the edges, the data is the *goaltype* already discussed. The nodes are either tactic-nodes, whose data is represented by a *reasoning technique* record, or goal-nodes. The reasoning technique record looks like this:

$$RTechn := \{\ name :: name\ ,\ appf :: app\_data\ \}$$

A tactic-node is evaluated according to the contents of the *app_data* field:

$$\textbf{\textit{type}}\ app\_data := Tactic\ name \mid Nested\ (name, \{Or, OrElse\})$$

This field contains the name of an atomic tactic (i.e. a tactic provided by the theorem prover), or the name of a graph tactic, which corresponds to a list of nested PSGraphs. In the latter case, a flag is used to indicate whether multiple nested graphs should be evaluated OR-style or ORELSE-style. The data on goal-nodes is given by the `gnode` type, described before.

We define a *PSGraph* as follows:

$$PSGraph := \{\ graph :: Graph,\ atomics :: name \xrightarrow{m} tactic,\ graph\_tactics :: name \xrightarrow{m} [\ Graph\ ]\ \}$$

The *graph* field contains the graph of the proof strategy. The *atomics* field is a map from names of atomic tactics to actual tactic functions provided by the prover. The *graph_tactics* field contains any nested graph tactics.

3.5 Combinators revisited

With the underlying PSGraph representation, we can provide more details about how the combinators are implemented. Since PSGraphs all contain their own lookup tables for atomic/graph tactics, it is important to avoid name clashes when we combine multiple graphs into one. To avoid such clashes we define a function type

$$\textbf{type}\ psgraph\_fun = PSGraph \to PSGraph$$

which is then used to combine PSGraphs. The idea is that given a PSGraph we generate fresh names when required, and this type enables us to linearise this generation by enforcing an order. The combinators then have types given by:

$$\begin{aligned}
\mathit{lift} &:: \mathit{name} \times \mathit{tactic} \times [\mathit{goaltype}] \times [\mathit{goaltype}] \to \mathit{psgraph_fun} \\
\mathit{THEN} &:: \mathit{psgraph_fun} \times \mathit{psgraph_fun} \to \mathit{psgraph_fun} \\
\mathit{TENSOR} &:: \mathit{psgraph_fun} \times \mathit{psgraph_fun} \to \mathit{psgraph_fun} \\
\mathit{REPEAT} &:: \mathit{psgraph_fun} \times \mathit{goaltype} \to \mathit{psgraph_fun} \\
\mathit{NEST} &:: \mathit{name} \times \mathit{psgraph_fun} \to \mathit{psgraph_fun} \\
\mathit{OR} &:: \mathit{name} \times \mathit{psgraph_fun} \times \mathit{psgraph_fun} \to \mathit{psgraph_fun} \\
\mathit{ORELSE} &:: \mathit{name} \times \mathit{psgraph_fun} \times \mathit{psgraph_fun} \to \mathit{psgraph_fun}
\end{aligned}$$

A *psgraph_fun* can be turned into a PSGraph by applying it to the empty graph:

$$\mathit{empty\_psgraph} :: \mathit{PSGraph}$$

3.6 The TP interface... OR how to connect a new theorem prover to Tinker

To connect a theorem prover, a structure implementing the *TP Config* signature must be provided. These structures for Isabelle and ProofPower are depicted in Figure 8 as *Isa_Tinker* and *PP_Tinker*. This structure contains all theorem prover related data types and functions that are required. Firstly, data types for the underlying terms, theorems, proof context and tactics must be provided[5]:

```
type term
type thm
type context
type tactic
```

A handful of common functions over these types must also be provided, such as matching of terms:

```
val match : context -> (term * term) -> bool
```

Tinker assumes that a (partial) proof will be of type `plan` and it contains a set of open goals of type `pnode`. A tactic is then seen as a function from a (partial) proof (`pplan`) and a goal (`pnode`) to a list of new goals and an updated proof. As there may be multiple branches when applying a tactic, a sequence of each branch is returned. This is captured by the `appf` type:

```
type pplan
type pnode
type appf = pnode * pplan -> (pnode list * pplan) Seq.seq
```

The user also needs to provide a means of producing `appf` functions from tactics, which are allowed to take a list of theorems as input:

```
val apply_tactic : thm list -> tactic -> appf
```

This interface was designed to provide the necessary functionality to interface with both Isabelle and ProofPower. We expect these to evolve when connecting more theorem provers in the future. Also note that so far both connected theorem provers are implemented in Poly/ML, making integration easier. For other platforms, integration may need to be done using e.g. socket communication.

[5]The list is not complete but gives an indication of the interface between Tinker and the underlying theorem prover.

3.7 Evaluating PSGraphs

Evaluation makes heavy use of both the prover and goal type functionality provided. A special record *Eval* is used to keep track of the evaluation state of a PSGraph G as it applied to a `pnode` p of a `plan` *prf*. *Eval* contains G itself, together with a list of branches of the evaluation. Each branch contains a stack of active graphs (to support hierarchical evaluation of PSGraphs) as well as *prf*.

In general, the graph G may have more than one input edge, so we create a branch for each input edge e by placing a goal-node containing p on e – as long as this goal-node matches the goal type of e.

Evaluation follows by picking a branch b and searching the PSGraph at the top of b's stack for a goal node g situated on an input edge of a tactic-node t. Firstly, g is removed from the graph. Then if t is an atomic tactic, it is applied to g, using the `apply_tactic` function. As a tactic may introduce branching, a sequence of updated `pplan`s coupled with a list *ps* of newly generated subgoals (`pnodes`), is returned. For each element in this sequence, the new subgoals are added to the output edge of t in a type-respecting way, without duplicating or loosing any of the subgoals. There may be multiple ways of achieving this, and each such way becomes a branch. It may also not be possible, which means that this element is discarded. To add a `pnode` to an edge, it must first be converted to a (serialisable) `gnode`. Such `gnode` is the return value of the `match` function of the goal type when it succeeds.

If t is a graph tactic, then g is moved to the corresponding input edge of the graph nested inside of t, and this graph is pushed on to the stack of active PSGraphs. A graph evaluation has successfully terminated when all goal nodes are on the output edges of the graph. If this is the only graph on the stack then the evaluation is complete. Otherwise, the top graph is popped from the stack and its output goals are added back to the parent graph on the appropriate edges.

Graph tactics nesting multiple graphs via *OR* or *ORELSE* are evaluated similarly, except branches are created for each subgraph, which are all evaluated (for *OR*) or evaluated until one branch completes successfully (for *ORELSE*).

4 Conclusion, related & future work

In this paper we have extended [7], which introduced the PSGraph formalisation, with details of how PSGraphs have been implemented in the Tinker tool – with support for the Isabelle and ProofPower theorem provers.

There are other tools supporting graphical representation of proofs. LΩUI [13] and XIsabelle [12] enable a graphical view of proof trees. They deviate from Tinker by viewing the proof and not the underlying proof strategies, which requires e.g. loops. Moreover, they cannot be used for inspection of the goal flow during evaluation, and crucially, the do not have the necessary goal abstraction via goal types. Moreover, tools such as ProveEasy [3] and Jape [1] have been developed to help students to learn how to do formal proofs, and deviates from Tinker in the same way as XIsabelle and LΩUI. Finally, as with XBarnacle[10], Tinker supports hierarchies, thus one can view a proof (strategy) at different levels of abstraction. Incorporating other features from XBarnacle is future work.

This is the first version of the tool, and we have already started the work on the next version, and we will now summarise features we would like to include in the next version. Firstly, goal types are a topic of active research, both from a theoretical perspective and from the point of view of implementation. The tool has therefore been developed to be flexible in the sense we can easily plug in new goal types. Experiments so far have shown that the predicate-style types described in [7] are easy to work with, but do not allow certain desirable features. On the other hand, the more expressive types from [8] are very powerful, but complex to work with.

Therefore, we are now focusing on defining a new goal type combining the simplicity of [7] with the power of [8], extended with variables and pattern matching as in e.g. Ltac [4].

In the next version of Tinker, we would like to integrate the evaluation and development views of the GUI to enable changing graphs during evaluation. The GUI should also show the proof state, and offer an improved layout mechanism whereby only goal-nodes change position during the course of evaluation. Moreover, more user control during evaluation is desirable, in particular allowing users to easily select which goal should be executed next, and combine 'interactive' and 'automatic' modes, where the user can choose to 'step over' or 'step into' hierarchies. Continuing with this 'debugger' paradigm, we would also like to add breakpoints in order to start 'interactive' mode at a particular step during an otherwise 'automatic' evaluation. This will be very useful for debugging large tactics such as SuperTac [11].

For the building aspect, we would like to make it easier to fold/unfold graphs when building new graphs, combined with a distributed way of defining search and evaluation strategies, where each nested graph may have its own strategy – configurable in the GUI. We would also like to add 'Library' functionality, as in e.g. Simulink[6], where a user can drag-and-drop existing, pre-made strategies into the current graph or create new Library entries from re-usable graph components.

Finally, we plan to integrate Tinker with more theorem provers. HOL4 should provide fewest obstacles as it is close to ProofPower and written in PolyML. Other provers, such as HOL light and Rodin, will require socket communication between Tinker and the prover core.

References

[1] Richard Bornat & Bernard Sufrin (1997): *Jape: A calculator for animating proof-on-paper*. In: *CADE-14*, Springer, pp. 412–415, doi:10.1007/3-540-63104-6_41.

[2] A. Bundy, D. Basin, D. Hutter & A. Ireland (2005): *Rippling: Meta-level Guidance for Mathematical Reasoning*. Cambridge University Press, doi:10.1017/CBO9780511543326.

[3] R. Burstall (2000): *ProveEasy: Helping people learn to do proofs*. *ENTCS* 31(0), pp. 16 – 32, doi:10.1016/S1571-0661(05)80327-5.

[4] David Delahaye (2002): *A Proof Dedicated Meta-Language*. *ENTCS* 70(2), pp. 96–109, doi:10.1016/S1571-0661(04)80508-5.

[5] Lucas Dixon & Jacques D. Fleuriot (2003): *IsaPlanner: A Prototype Proof Planner in Isabelle*. In: *CADE-19*, *LNCS* 2741, Springer, pp. 279–283, doi:10.1007/978-3-540-45085-6_22.

[6] Lucas Dixon & Aleks Kissinger (2010): *Open Graphs and Monoidal Theories*. *CoRR* abs/1011.4114.

[7] G. Grov, A. Kissinger & Y. Lin (2013): *A Graphical Language for Proof Strategies*. In: *LPAR*, *LNCS* 8312, Springer Berlin Heidelberg, pp. 324–339, doi:10.1007/978-3-642-45221-5_23.

[8] Gudmund Grov & Ewen Maclean (2013): *Towards Automated Proof Strategy Generalisation*. *CoRR* abs/1303.2975. Available at `http://arxiv.org/abs/1303.2975`.

[9] A. Kissinger, A. Merry, L. Dixon, R. Duncan, M. Soloviev & B. Frot (2011): *Quantomatic*. https://sites.google.com/site/quantomatic/.

[10] Helen Lowe & David Duncan (1997): *XBarnacle: Making Theorem Provers More Accessible*. In: *CADE1-4*, Springer-Verlag, pp. 404–408, doi:10.1007/3-540-63104-6_39.

[11] Colin O'Halloran (2013): *Automated verification of code automatically generated from Simulink*. *Automated Software Engineering* 20(2), pp. 237–264, doi:10.1007/s10515-012-0116-5.

[12] Maris A Ozols, Anthony Cant & Katherine A Eastaughffe (1997): *XIsabelle: A system description*. In: *CADE-14*, Springer, pp. 400–403, doi:10.1007/3-540-63104-6_38.

[13] Jörg H. Siekmann, Stephan M. Hess, Christoph Benzmüller, Lassaad Cheikhrouhou, Armin Fiedler, Helmut Horacek, Michael Kohlhase, Karsten Konrad, Andreas Meier, Erica Melis, Martin Pollet & Volker Sorge (1999): *LOUI: Lovely OMEGA User Interface*. *Formal Asp. Comput* 11(3), pp. 326–342, doi:10.1007/s001650050053.

[6] See `www.mathworks.co.uk/products/simulink/`.