7 LogTell-R Architecture

7.1.Chapter Preface

The main goal of this article is to provide an overview of the LogTell-R system, the considerations taken into account when developing it, and the responsibilities of its different modules. Along with the description of the system components, three usage scenarios are described to highlight some of the system's main features from a user point of view.

This text was initially written to appear as a standalone technical report and, later, as part of a paper on LogTell-R's extensions to fully implement **player mode** interaction. It remained an unpublished manuscript prior to this thesis.

7.2.Introduction

A software architecture should lend itself to incremental implementation via the creation of a "skeletal" system in which the communication paths can be exercised, but which at first has minimal functionality [Bass et al. 2003]. This skeletal system can then be used to develop the system gradually.

The high-level architecture of LogTell-R attempts to provide a good separation of concerns between the modules involved in the problem of story craft [Karlsson et al. 2009b], while keeping in mind the possible addition, in the near future, of player interaction during dramatization. We believe that this ultimate goal should be attainable, since it is within the reach of the extended conceptual model (and in view of the facilities implemented to support it), with minor changes to the system.

The process of developing this infrastructure was guided by four overarching considerations:

- Separation of concerns between the different sub-problems of story craft;
- Maximum opportunities for interaction as player, with minimal changes to the author-mode architecture;
- Software maintainability and scalability to address future needs; and
- Authorability with respect to content.

7.3.Architecture Overview

The four major modules of LogTell-R are: Knowledge Base Editor, Context Control Module (CCM), Plot Manager (PM), and Drama Manager (DM). Basically their responsibilities are divided as following: our proof-of-concept editor allows the creation of pieces of data to be stored in the knowledge base (KB) repository; CCM controls access to the KB and to the plot model; PM deals with *plot-level* decisions; and DM deals with the narration of stories and the presentation decisions (*story-level* and *text-level*).

Although many works that mix storytelling and game features sometimes refer to plot managers as drama managers [Roberts and Isbell 2007], we chose to preserve the distinction between the two, following the breakdown of the story craft problem into its more manageable sub-parts presented in [Karlsson et al. 2009b].

We also pay attention, wherever possible, to issues related to the development of tools for non-technical users. Content, in particular, is a key point in movies and games – and so it is in any Interactive Storytelling application. However, we do not claim that our tools have reached a satisfactory degree of usability as of yet.

By refactoring the system to better separate concerns between the different modules, adding a bi-directional communication channel between the Drama Manager and the Plot Manager, and trying to apply simple integration patterns, we achieve a more flexible architecture which, with few extensions, can support both author and player interaction.

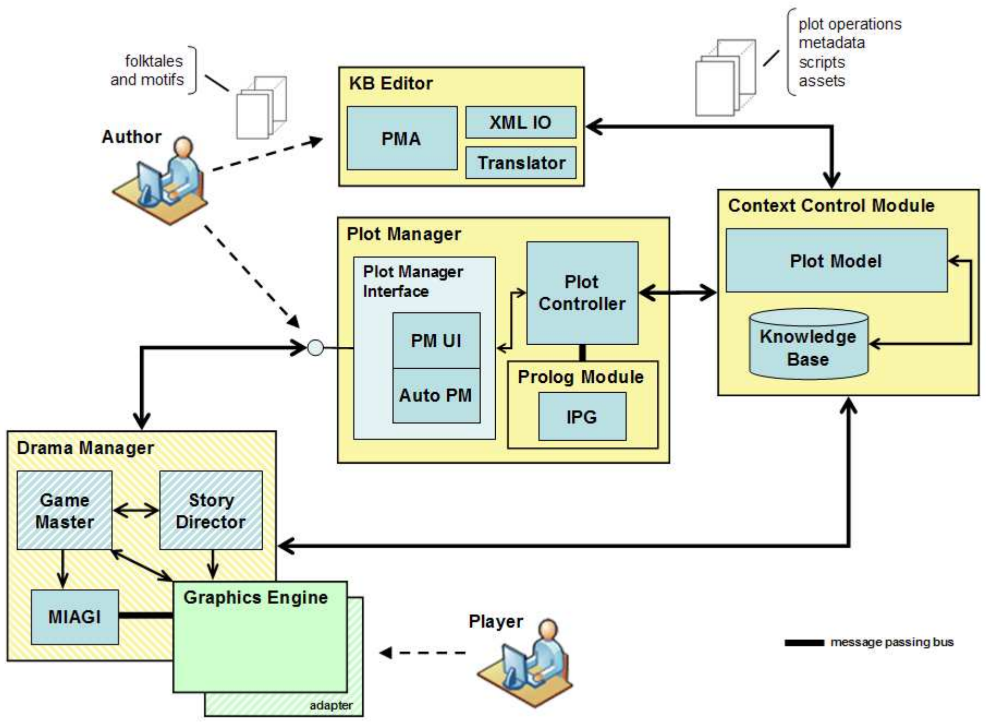


Figure 7.1: LogTell-R architecture overview and user roles.

The use of LogTell-R is basically divided in two stages:

- Creation/manipulation of the story space context (conceptual model and additional data); and
- Usage of Plot Manager over the defined context to create stories, interactively.

The Plot Manager is intended for use directly through its user interface (UI) in **author mode** or programmatically by Drama Manager in **player mode**. A diagram of the architecture is shown in Figure 7.1. The hashed areas under Drama Manager are not currently fully implemented.

7.3.1.Knowledge Base Editor

The knowledge base comprises all the available knowledge data, not necessarily stored in the same place or available to every program module. Ideally, any sort of manipulation of the knowledge base should be properly assisted by authoring tools, in order to help prospective authors in creating the system story space with as little effort as possible. This is important to make the underlying complexity of the knowledge base transparent to authors, especially if complex logic models or specialized programming languages are utilized.

For our current prototype, we created a proof-of-concept editor to be used when performing some tasks over the KB. The main goal of this editor is to allow an author to experiment with the conceptual model and to construct a hierarchy of typical plans.

The creation of the hierarchy makes use of PMA [Karlsson et al. 2009a], supported by a simple planner to check pre- and post- conditions of operations, and some template-based facilities to express plot events in textual format. This allows one to more easily try and experiment with different chains of events, before generating the hierarchy of typical plans that will later be used during the composition and adaptation of plots.

The user can at any time visualize the created hierarchy. The user interface of the editor shows the list of plot events in the dynamic scheme, a list of PMA operations, and the describing PMA formula defined so far.

Moreover, when an algebra operation is performed, the editor may query the user for metadata, in order, for example, to give a name to a new complex operation resulting from the **union** of two alternatives, or to indicate what fact, relationship or attribute will determine the removal of a sequence of events via the **difference** operation.

The editor also allows editing operations in the dynamic schema of the conceptual model, primarily in regard to their pre- and post- conditions (in turn, based on the static schema), but also with respect to composition relations between operations.

In the case of the special operations that represent motifs, the editor also asks the user to associate the motif operation with Lua scripts, to be used in controlling the behaviour of each character involved in its dramatization.

Finally, the editor translates the edited context data into a XML representation and creates the necessary Prolog files.

7.3.2.Prolog Module

The Prolog Module is exclusively accessed by the Plot Manager, given that no other module in the system should make direct use of the Interactive Plot Generator (IPG).

The Prolog module encapsulates IPG instances and translates the Prolog clauses into a more abstract Plot Model, which is used by the higher-level system elements. In this way we avoid passing Prolog clauses around and concentrate the effort of parsing responses from the Prolog engine in only one place.

One additional benefit is that, if the specific Prolog implementation is changed, maintenance impacts are restricted to the Prolog Module. To enhance this loose coupling, all communications between the Prolog Module and the Plot Manager happen via a message passing system (to be described later).

Handling the abstract Plot Model is a responsibility of CCM, as other modules may also need access to it. Another responsibility of the Prolog Module is handling the plan recognition routines. In doing so, it behaves in the same way as when dealing with plan generation.

It is interesting to note that, while recent efforts exist [Ramírez and Geffner 2009] to narrow the gap between plan recognition and planning — treating plan-recognition as a planning problem —, work in plan-recognition proceeds using mostly handcrafted libraries or algorithms that are not directly related to planning (significant examples can be seen in [Kautz and Allen, 1986; Lesh and Etzioni, 1995; Avrahami-Zilberbrand and Kaminka, 2005]). Our approach, as available in IPG, is also based on plan libraries (specifically [Kautz 1991]).

7.3.3.Context Control Module (CCM)

The CCM implements a repository-style facility [Garlan and Shaw 1994], with two kinds of components: a central data repository, and a collection of independent components to operate on this data store.

CCM encapsulates the KB and is responsible for providing defined interfaces to access its data. Among other items, it keeps the Plot Model, an abstraction from our lower level representation of plots in Prolog.

As IPG needs a Prolog representation of the genre context during runtime, Plot Manager (PM) queries CCM for it, which in turn and passes the context back to the Prolog Module. Because of this duplication, in order to maintain consistency between IPG's and CCM's views of the storyworld, every time IPG performs a planning step the CCM is notified (by PM) about the changes affecting the current facts. Similarly, if some event happens in the storyworld and IPG is not aware of it, the CCM will send the relevant information to be asserted into the Prolog Module.

We decided to cleanly separate plot model and controller logic for the sake of increased maintainability. As the Plot Manager is the only module that accesses IPG, being therefore the module directly in charge of building the plots, the controller logic is its responsibility. Since other modules need to query the Plot Model to perform their tasks, the plot model stays under CCM's responsibility.

Finally, CCM only deals with assets (models, textures, scene definitions) on the same level as LOGTELL's original Drama Manager, described in [Pozzer 2005]. Consequently, the interface for accessing this information needs to be changed in order to comply with the requirements of the new graphics engine (Unity 3D [Unity]).

7.3.4. Plot Manager

The Plot Manager (PM) can be seen as a tool for the creation of stories at the fabula level, and must guarantee that stories satisfy some specified constraints, such as those related to the conventions of a given genre. In LogTell-R, the same as in LOGTELL, the craft of plots is still pursued in a stepwise fashion.

It is important to note that the description of a storyline at the fabula level is always expected to look much simpler than at the story/text level. This is what justifies the use of a planning system, duly programmed to avoid any unnecessary overloading, to be judiciously intercalated with the author's direct participation.

Author interaction happens through the Plot Manager's user interface, and the overall plot composition process takes the form of successive cycles of goal-inference, planning, plan recognition, and user intervention. In addition, in order to allow the Drama Manager (DM) to also send commands to PM, we provide means of programmatically controlling plot generation. As a consequence, we designed LogTell-R's Plot Manager to operate in two ways: via a Graphical User Interface (GUI) or as an "automatic" process. Both implementations extend an abstract Plot Manager interface and use the same Plot Controller to manipulate the Plot Model.

In autonomous mode the process of composition continues by inferring new goals from the situations generated in the first stage, when notified by the DM. Another LOGTELL-related work [Camanho et al. 2009] has also pursued similar efforts with a continuous narration flow that uses multiple instances of IPG. Further implementation is necessary in our system to deal with more than one instance of IPG.

One limitation of the autonomous mode is that the event to be used next in the plot is randomly selected from among the available alternatives. This selection could certainly be improved by the use of adequate heuristics to evaluate the "interestingness" of using each event.

When using Plot Manager via its GUI, an author is allowed to also resort to plan-recognition during plot composition and adaptation (this feature is described in more detail in [Karlsson et al. 2006a, Karlsson et al. 2010a]).

Also, the GUI allows the insertion of special operations representing motifs in the plot to introduce shifts in the story context. PM loads the list of available motifs from the CCM and shows it to the author. When the author selects the motif to be inserted, a new item corresponding to it is loaded into the dynamic schema used by IPG. As an enhancement to the current implementation, it should be possible to annotate those special items with additional information, to signalize to IPG how they should be treated, distinctively from the regular operations.

In general, a Plot Manager might also have the ability to set restrictions that prepare for the narration. However, the final presentation order and timing of the events should be stipulated by the story narrator (as long as it does not violate any of the constraints imposed in the course of plot generation).

7.3.5.Message-passing Bus

As a mechanism to increase flexibility and avoid tightly coupled system elements, we make use of a message passing bus to integrate different services.

Event-based mechanisms such as this allow implicit invocation; one system element does not need to know about the internals of another. Any new element being integrated into a system can easily be introduced, simply by registering for the documented events generated by that system.

A second benefit is that implicit invocation eases system evolution [Sullivan and Notkin 1990], which is one of the goals of our architecture.

To implement this mechanism, two elements were added to our system: an event queue and a simple event processing engine. This same mechanism implementation has been applied between AI middleware and 3D engine in [Karlsson 2005].

7.3.6. Drama Manager

Basically the Drama Manager is the module responsible for the dramatization of a plot. In the original LOGTELL prototype it was responsible for dramatizing plot events and synchronizing characters' actions and the graphical representation. In doing so, the Drama Manager (DM) included an agent model implemented by hard-coded state machines. DM features also included a set of camera-placement techniques and an in-house built graphics engine.

One of our goals here was to decouple the Drama Manager sub-components to ease maintenance and to facilitate the addition of new features to enrich plot dramatization.

As the Drama Manager is responsible for story visualization, it should focus on story-level decisions, leaving media-specific decisions or low-level concerns to sub-modules.

We break down DM into three auxiliary sub-modules and one external element:

- Story Director;
- Game Master;
- AI middleware; and
- Graphics engine (external).

The Drama Manager is then responsible for dealing with how to transform plot (at Bal's *fabula* level) into narration (Bal's *story* level) and for coordinating AI, graphics, storyworld management, camera placement, etc.

As described in [Karlsson et al. 2009b], it is also part of the role of the story narrator to define the degree of freedom that the player will have when interacting with the story. When dealing with audience interaction, for instance, it is important to establish that any change to the plot is performed by the Plot Manager. If any unplanned change happens, DM must interact with PM. This way, the DM can be seen as a mediator between the plot generation engine and the audience, which allows for great flexibility in handling different approaches for audience interactivity.

The **Story Director** deals primarily with conventions of exhibition media. The **Game Master** deals with storyworld management. In order to make it more flexible, lower-level character behaviours are moved to a separate **AI middleware** under its control. The **graphics engine** handles physics and visuals (effects, particle systems, etc.), and possibly the player user-interface.

It is only natural to use conventions from cinema (pacing, establishing shots, etc.) in dramatizing a plot in a 3D world, especially for a passive audience.

Our current proposal for this area could also be integrated with the cinematography director in [Lima et al. 2009]; whose responsibilities are divided into Scriptwriter, Scenographer, Director, and Cameraman. The scriptwriter in our system is the Drama Manager itself, dealing with how to present a story; and the scenographer is the Game Master, as it is the module responsible for instantiating agents and managing the Storyworld and its entities. The Story Director can centralize the cinematography knowledge and is responsible for deciding how to best present story events (scenes). We can use the idea of a coordinating Cameraman, who instantiates different cameras and waits for director's input.

Specific motifs can also act as dramatization tools, to solve the problem of how to narrate specific events. One example is *life token*, where an object aspect changes appearance, thus solving the problem of how the hero could learn that the princess was in distress.

As briefly mentioned, the Game Master is responsible for controlling certain aspects of the storyworld, relieving the load usually imposed on the Drama Manager. Consequently it also coordinates with the AI and graphics engines.

As stated by Cook [2009] when talking about tabletop RPGs, the GM works with players to create the story, adding random encounters, creating and controlling minor characters (extras) in the story – even animals and monsters – and generally behaving as improvisational actor. The GM can also take care of the game/world internal mechanics, dealing with rules, combat outcomes, results of actions, etc.

The PM can totally ignore these extras, since only as groups they have some influence over the plot conduction.

The degree of autonomy granted to extras leaves them free to perform certain actions, such as walking in the game world pursuing their own business. When required to participate in some plot event, which has always a higher priority, the Game Master (through, for example, an auxiliary AI middleware) makes them interrupt momentarily whatever they were doing.

One such AI middleware that fits our architecture is MIAGI [Karlsson 2005]. It offers a series of facilities (such as an agent model; different options for character behaviour implementations: simple goal-planning, FSMs, and fuzzy rules; scripting in Lua; layered character movement features) that can be used to control low-level behaviours of the main characters and to give more life to the environment by providing autonomous supporting characters.

MIAGI's agent model includes a plug-in mechanism for easily integrating new ways to implement character behaviours. The plug-in structure allows one to easily replace implementation of any behaviour component or to add alternative implementations.

Another advantage of using a middleware component is to help alleviate part of the burden within the production pipeline. Middleware provides a baseline of tools that support the growing complexities of development [Meloni 2009], thus allowing resources to be focused on content quality.

The proposed architecture also allows extending the GM module to assume some additional responsibilities in future versions, such as:

- Act as a semi-autonomous mediator between players and the Drama Manager, in the role of a tabletop RPG-inspired agent, solving some interaction issues via heuristics [Laws 2001]. Ongoing efforts to quantify and categorize these repertoires of "rules" could be very useful in further pursuing this direction;
- Respond to events happening in the environment (via triggers associated with certain actions or areas in the game world); and
- Use plan-recognition against plan libraries to try and predict the goals being pursued by players and try to adapt to them. While usually a non-trivial problem in general, plan-recognition is particularly interesting in a system that uses a plot model. The dramatic structure of a plot greatly simplifies the problem of determining a user/player's plan and if it might interfere with some plot event.

7.4. System Usage Scenarios

In order to briefly illustrate the use of LogTell-R core functionalities in author mode, we shall describe here its three main usage scenarios:

1- Interacting with LogTell-R's Plot Manager UI using its plan generation/recognition facilities as support in creating a plot;

2- Interacting with LogTell-R's Plot Manager UI inserting motifs at the appropriate point in a plot being created; and

3- Using the knowledge base editor to create the hierarchy of typical plans to be used as support when authoring plots.

Even though the knowledge base editor is only a proof-of-concept implementation and its UI needs a thorough redesign, its implementation helps envision the usage of PMA [Karlsson et al. 2009a] in LogTell-R and brought valuable insight into the requirements for such editing tools.

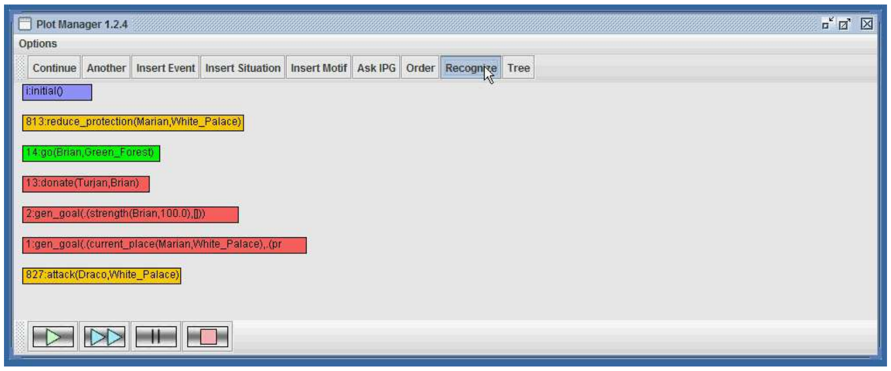
In all three scenarios presented here we make use of the same conceptual model specification of a Swords & Dragons genre. A description of the logic context for this model can be found in [Karlsson et al. 2006a] and [Karlsson and Furtado 2010a].

An example plot, as generated by LogTell-R over this genre, tells the classical happy-ending story: "Marian, the princess, dismisses some of her guards, causing the protection of her castle to be reduced. Draco, the dragon, regards that as an opportunity to kidnap her. Draco then goes to Marian's Castle, attacks the castle and kidnaps Marian. As a noble knight, Brian feels compelled to save her. But, before that, he needs to ask for Turjan's magic to increase his strength. He then goes to Draco's Castle, attacks the castle and fights Draco. He kills Draco and frees Marian, who starts loving her saviour. Motivated by their mutual affection, Brian and Marian go to the church and marry each other. And they live happily ever after."

7.4.1.Using Plan Generation / Recognition

One possible way to start composing this plot is to ask the planning system to start generating a plot from the initial state of the world. According to the goal-inference rules in the behavioural schema of the conceptual model, this would possibly cause the princess to dismiss the guards, reducing the protection of the castle. Draco will then attack the weakened castle and Brian will ask the magician for strength (as shown in the plot generated so far in Figure 7.2).

The author of the story could go on using only the plan generation features of the system (as in LOGTELL [Ciarlini et al. 2005]), but let’s assume that he is not sure about how to proceed to create an interesting story.



```
i:initial()
813:reduce_protection(Marian,White_Palace)
14:go(Brian,Green_Forest)
13:donate(Turjan,Brian)
2:gen_goal(.(strength(Brian,100.0),[]))
1:gen_goal(.(current_place(Marian,White_Palace),.(pr
827:attack(Draco,White_Palace)
```

Figure 7.2: Plot Manager showing selected events for plan recognition.

In LogTell-R the author can ask the system to use plan recognition against a library of typical plots to suggest events to be included in the plot being composed. The author can select events from the plot composed so far and ask the system for “plot pieces” (or complex events) that include the selected events. For example, the author selects the *reduce_protection* and *attack* events (shown in orange in Figure 7.2) and issues the **recognize** command by clicking on the appropriate button in Plot Manager's UI.

Plot Manager will then show the Hierarchy of Typical Plans in a separate window with the higher-level event containing the selected events marked in red and the remaining component events in orange. In our example, the system will identify that the selected events can be part of the higher-level *abduct* event, marked in red (which in turn is an event of the type *do_villainy*, also in red) and that the *kidnap* event, marked in orange, is the missing event to complete the *abduct* event. This result can be seen in Figure 7.3; blue edges denote `part_of` relations and red edges denote `is_a` relations among events in the hierarchy.

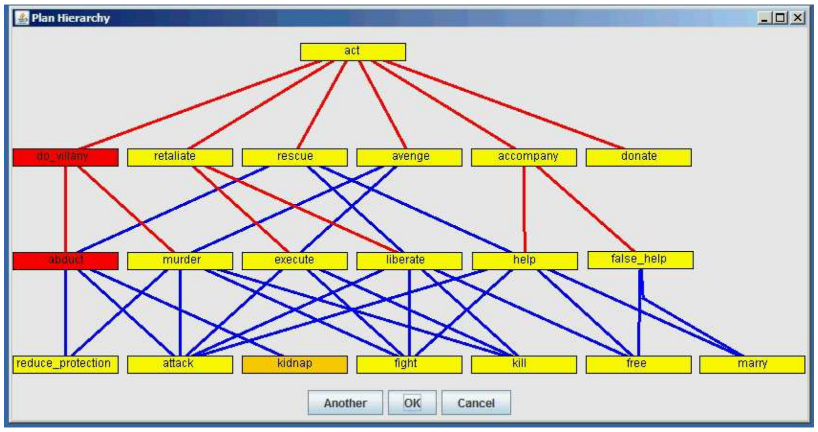


Figure 7.3: Plan hierarchy window in LogTell-R with highlighted events.

After deciding that this is the plot piece he wants, the author can click OK and the system will include the events marked in orange into the plot. In our example, the kidnap *event* will be added to the plot. The author can go on making alternate use of plan generation, plan recognition, and manual interference steps in composing the plot until its happy ending.

7.4.2.Using Motifs

In order to illustrate LogTell-R's support for antithetic relations, let's branch from the previously described story when Brian attacks the dragon castle.

In our first example the knight attacks the dragon's castle and kills the dragon. The story author could decide, instead, that Brian should not go alone and that his helper – Hoel - is the one that manages to kill the dragon and free the princess (as shown in Figure 7.4).

As we know from the genre conceptual model, if the victim of the kidnapping is freed, she'll fall in love with her savior. Thus if Hoel frees Marian, she'll fall in love for him, not for Brian.

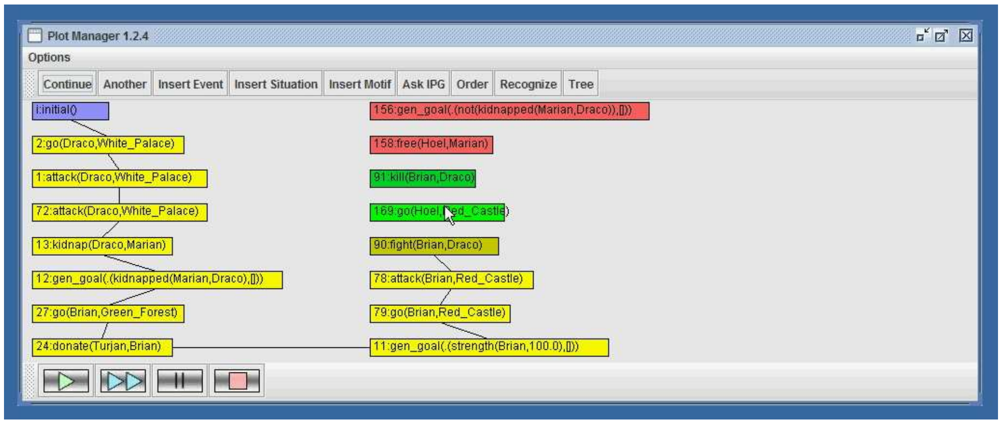
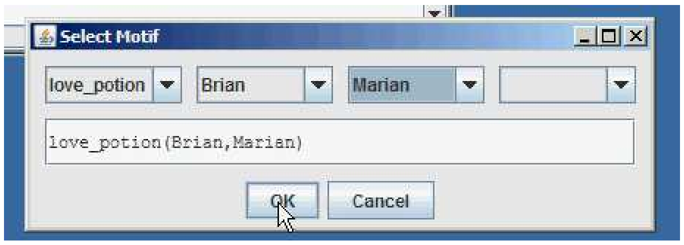


Figure 7.4: Example generated story where two knights help save the princess.

But Brian is the main character in the story so far. If the story author wants him to be part of the happy ending with Marian, we need some means to reconcile their amorous attachments.

For such cases, involving the addition of events that would not normally follow from the preceding ones, LogTell-R allows the selection of a motif (from a pre-defined set) to be inserted into the appropriate spot, thus making the sequence possible. In the described case, the *love_potion* motif could be used to make one character fall in love for the other.

The author can then use the **Insert Motif** command (through the dialog shown in Figure 7.5) to strongly interfere with the plot, inserting the special event where needed.



love_potion(Brian,Marian)

Figure7.5: “Insert Motif” dialog.

After inserting a motif, the author asks the system to validate the insertion (**continue** command) and goes on with the creation of the plot via plan generation/recognition.

In our example, after Hoel frees the princess, Brian gives her a potion that makes her fall in love for him. Due to their new mutual affection, Brian and Marian go to the church and get married. This sequence of events is shown in Figures 7.6 and 7.7.

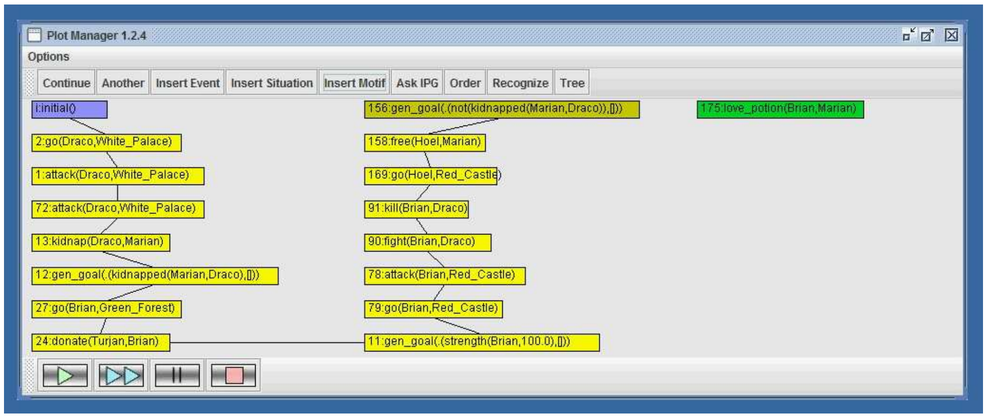


Figure 7.6: Example story with the insertion of a special event (motif).

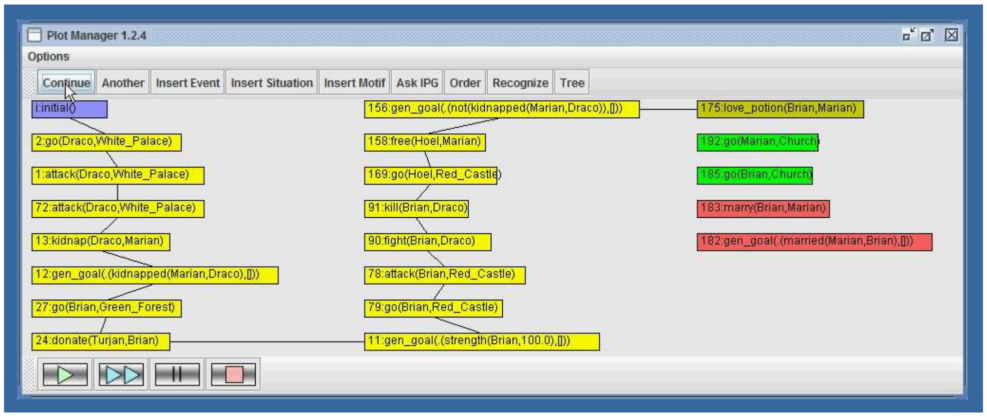


Figure 7.7: Example story containing motif.

7.4.3.Creating the Hierarchy of Typical Plans

While the knowledge base editor allows editing the schemas and operations in the conceptual model of the genre, its main use case is allowing the creation and exploration of the story space encompassed/defined by the genre.

LogTell-R's approach is that the conceptual model of a genre represents the story space, and that a hierarchy of typical plots within that genre definition helps visualize the structure of the possible stories.

A proof-of-concept editor was developed so that the "story space author" can more easily experiment with different ways to chain the events in the model, and use this knowledge to revise the specification so as to improve the system's story space. During this exploration, the hierarchy of typical plans is built.

The core of the editing tool makes use of Plot Manipulation Algebra (PMA) [Karlsson et al. 2009a]. By applying the algebra operations to the set of events in the conceptual model, the user can play with the story space and check what kinds of stories are being generated.

The editor main UI is divided into four areas (as shown in Figure 7.8). The topmost area contains a toolbar with a button for each PMA operation. The centre-left area shows the available events in the conceptual model (both simple and complex events). The centre-right area shows the algebra expression currently under scrutiny. The bottom part of the screen shows the results of evaluating the expression in the context of the conceptual specification of genre. These results are show both in Prolog notation and as plain text to facilitate user understanding of the possible stories being generated.

The interaction with the prototype hinges on how the PMA expression is manipulated. Starting from the empty plot [], the user can apply algebra operations to it (possibly involving events in the model). At any moment, the user can query the system for the possible outcomes of the plot-expression.

Here we shall briefly sketch how the creation of the hierarchy can happen in the editor. A 'partial' account of the construction of the structure shown in Figure 7.3 will serve as example in a series of steps.

1- As mentioned, the expression initially represents the empty plot – *[]*

2- User adds an abduction to the plot space by applying the product operation:

   a. user clicks on the expression node for *[]*;

   b. user clicks on the **product** button in the toolbar;

   c. then he clicks on the button for the *abduct* operation in the left pane;

   d. as the product of the empty plot with an event is the event itself, the resulting expression is: *abduct*

3- To produce a sequence where abduction is followed by liberation:

   a. user clicks on the node for *abduct* in the expression (in this case, the whole expression);

   b. user clicks on the **product** button in the toolbar;

   c. he then clicks on the button for the *liberate* operation in the left pane;

   d. the resulting expression is: *(abduct * liberate)*

4- As liberate and execute correspond to the same paradigm after an abduction, the user adds this generalization to the expression:

   a. user clicks on the node for *liberate* in the expression;

   b. user clicks on the **union** button in the toolbar;

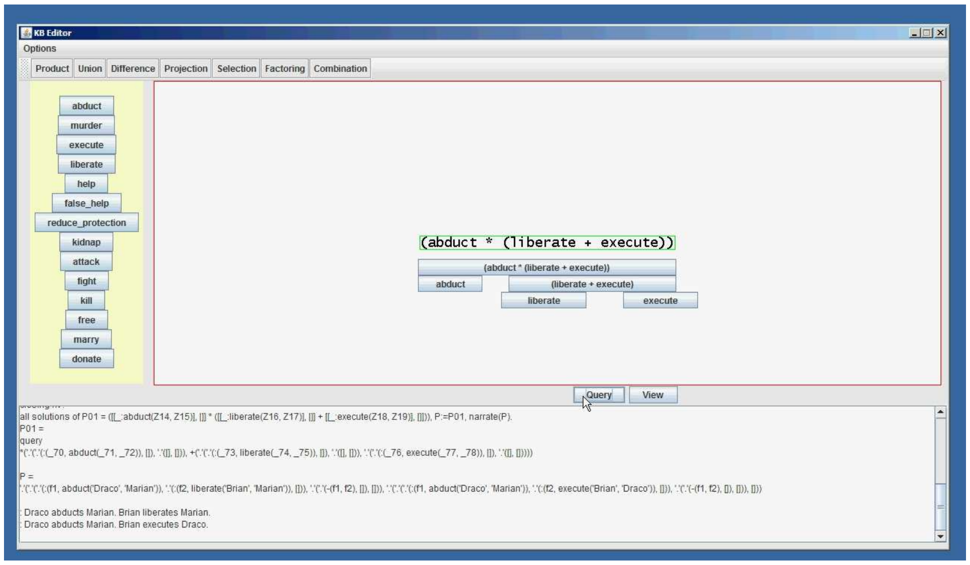
   c. user clicks on the button for the *execute* operation in the left pane;

d. the system asks the user to name this generalization ("retaliation", for example) and stores this information as metadata associated with the expression;

e. the resulting expression becomes: *(abduct * (liberate + execute))*

5- The user can query the underlying system to check the possible stories being generated by clicking on the **Query** button. Figure 7.8 shows the process so far.

6- The process goes on until the user is satisfied with the story possibilities defined by the PMA expression.

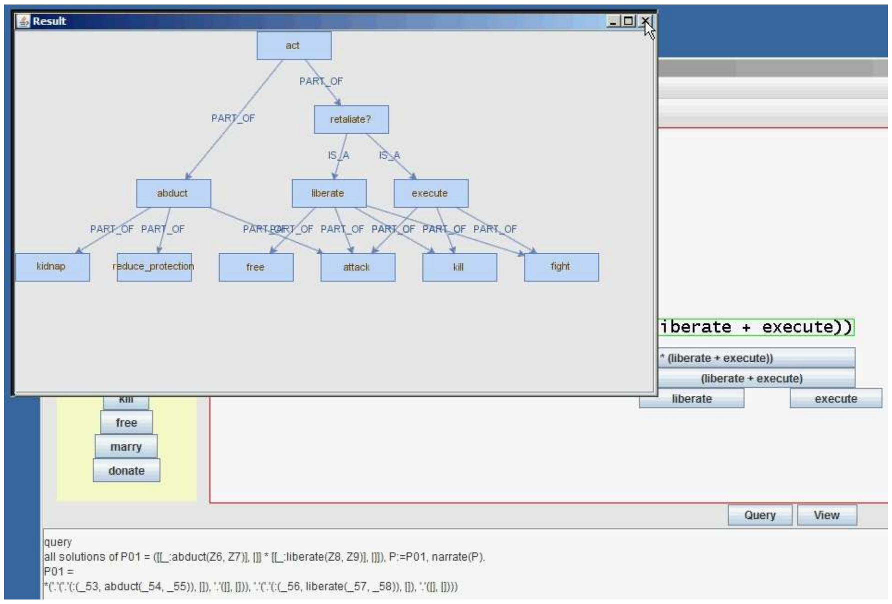


```
all solutions of P01 = ([[_:abduct(Z14, Z15)], []] * ([[_:liberate(Z16, Z17)], []] + [[_:execute(Z18, Z19)], []])), P:=P01, narrate(P).
P01 =
query
*('.'('.'(:(_70, abduct(_71, _72)), []), '.'([], [])), +('.'('.'(:(_73, liberate(_74, _75)), []), '.'([], [])), '.'('.'(:(_76, execute(_77, _78)), []), '.'([], []))))

P =
'.'('.'('.'(:(f1, abduct('Draco', 'Marian')), '.'(:(f2, liberate('Brian', 'Marian')), [])), '.'('.'(-(f1, f2), []), [])), '.'('.'('.'(:(f1, abduct('Draco', 'Marian')), '.'(:(f2, execute('Brian', 'Draco')), [])), '.'('.'(-(f1, f2), []), [])), []))

: Draco abducts Marian. Brian liberates Marian.
: Draco abducts Marian. Brian executes Draco.
```

Figure 7.8: Knowledge Base Editor main UI view.



```
query
all solutions of P01 = ([[_:abduct(Z6, Z7)], []] * [[_:liberate(Z8, Z9)], []]), P:=P01, narrate(P).
P01 =
*('.'('.'(:(_53, abduct(_54, _55)), []), '.'([], [])), '.'('.'(:(_56, liberate(_57, _58)), []), '.'([], [])))
```

Figure 7.9: Typical plan hierarchy as seen during its construction in the editor.

At any moment during the interaction with the plot expression, in the proof-of-concept editor UI, the user can also visualize the resulting hierarchy of typical plans by clicking on the **View** button. The hierarchy generated so far is then shown in its own window, as shown in Figure 7.9.

7.5. Final Remarks

In LogTell-R we strived both to augment the expressiveness of the narrative model underlying the system and to provide better tools to prospective authors intent on creating and telling interactive stories. To accomplish these goals, we focused on enhancing the flexibility of the story space spanned by the system, primarily with regard to the addition of support for handling meronymic and antithetic relations between events in the plot.

By refactoring the system to better separate concerns between its different modules, while keeping authorability (with respect to content) in mind during development and also maximizing opportunities for the implementation of player interaction (with as minimal as possible changes to the author-mode architecture), we achieve a more flexible architecture which can be used as a base to further explore the storycraft problem [Karlsson et al. 2009b] and which also makes it easier to compare different approaches to its sub-problems.

Having authorial tools in place to render the notational complexities of the knowledge base transparent to authors is essential for any system that aims to reach real world usage. The primarily objective of such tools is to reduce the burden on authors using the system, but they should also guide them in the creation of interesting stories. Creating an interactive environment that behaves as expected can be a tiresome task, as it's done now mostly by trial and error. Accordingly, we developed a proof-of-concept editor of the knowledge base to address some of these issues.

The main purpose of the editor is to help exploring the story space and to construct the hierarchy of typical plans to be used during plot composition. The knowledge base editor makes use of PMA (supported by a simpler planner to check pre- and post- conditions of operations) to allow one to experiment with different ways to chain the events in plots, and to use this knowledge to revise the specification so as to improve the story space.

While our prototype tool to edit the knowledge base still leaves much to be improved, we feel it brought important insight, and we now have a much better grasp on the priorities for future work.

Although the process of plot composition and adaptation in LogTell-R could surely be enriched far beyond what is currently supported, what we managed to accomplish seems to provide a sound basis to treat, at least, genres that exhibit a high degree of regularity.

Lastly, much work remains to be done towards the complete implementation of the player mode of interaction especially regarding the Story Director and Game Master modules.