An Incremental Algorithm for High-Performance Runtime Model Consistency

Christopher Wolfe[1], T.C. Nicholas Graham[1], and W. Greg Phillips[2]

[1] School of Computing, Queen's University, Kingston, Ontario, Canada
(wolfe|graham)@cs.queensu.ca
[2] Department of Electrical and Computer Engineering, Royal Military College of Canada, Kingston, Ontario, Canada
greg.phillips@rmc.ca

Abstract. We present a novel technique for applying two-level runtime models to distributed systems. Our approach uses graph rewriting rules to transform a high-level source model into one of many possible target models. When either model is changed at runtime, the transformation is incrementally updated. We describe the theory underlying our approach, and show restrictions sufficient for a simple and efficient implementation. We demonstrate this implementation in *Fiia.Net*, our model-based toolkit for developing adaptive groupware. Developers using *Fiia.Net* control components and connections through a high-level conceptual runtime model. Meanwhile, the toolkit transparently maintains the underlying distributed system, and propagates failures back into the conceptual model. This approach provides high stability, and performance that is sufficiently fast for interactive applications.

1 Introduction

Recent years have seen a proliferation of computing devices, ranging from smart telephones and PDAs, to netbooks and tablet PCs. When connected over a network, these devices enable new styles of communication and collaboration in mobile settings. Applications include meetings at a distance [1], tele-health [2] and online games [3]. Such adaptive groupware is fundamentally a distributed system which undergoes significant runtime adaptation: as users move between tasks, roles, devices and locations; and as network conditions and connections change. An example application might allow two software developers to brainstorm using a large shared touch-screen. They send a snapshot to a colleague's smartphone, who then migrates it to a laptop and joins the live collaboration. These systems are difficult to build, because they must provide intuitive user interfaces, while maintaining high performance through varying user demands and partial failure.

Model-based techniques have great potential for aiding the development of adaptive groupware. High-level conceptual models [4–7] can describe the system's structure, abstracting low-level issues like data sharing and caching policies, concurrency control algorithms, and network protocols. Distribution models can help reason about architecture trade-offs [8] or configure an implementation [9].

To address the challenges of adaptive groupware, we have developed a model-based system which supports runtime adaptation in both conceptual and distribution models:

- The runtime system automatically refines the conceptual model into a distribution model. This mechanism supports multiple possible implementations of each conceptual model.
- Developers specify high-level changes as runtime adaptations to the conceptual model (e.g., a user changes device, or new data is shared between users). The runtime system propagates these adaptations through refinement to the distribution model.
- The underlying distributed system takes its configuration from the distribution model, and reports failures as distribution adaptations (e.g., a smartphone's battery dies, or a network becomes unavailable). Following failure, the runtime system restores the models to a consistent state, allowing the application to detect and manage failures via the conceptual model.

This approach requires us to maintain consistency between the two models at runtime. Existing model transformation techniques, outlined in Sec. 2, do not support bidirectionality, limit the flexibility of the transformation, or are too slow for use in a running groupware system.

Underpinning our solution is a novel algorithm for maintaining bidirectional model consistency. Using this algorithm, refinement from a conceptual model to a distribution model is specified using unordered graph rewriting rules. Both models are maintained at runtime. Arbitrary changes in the conceptual model and removals from the distribution model are rapidly propagated through the transformation. The algorithm performs adaptations incrementally, and with minimal change to the models.

Our algorithm is possible due to restrictions in the scope of distribution adaptations. Changes in the distribution level result from failure, so are always reported via removals. This avoids the need to reverse-engineer new distribution model elements, so permits extremely general rules and high performance. Our algorithm does not otherwise depend on the behavior of distributed systems, and is independent of our particular metamodels and rewriting rules.

We have used this algorithm in the *Fiia.Net* groupware development toolkit [10]. *Fiia.Net* has been used to develop a range of applications, distributed across desktop PCs, smartphones, and tabletop computers. Significant examples include a game sketching tool [11] and a furniture layout application [10].

This paper is organized as follows. After reviewing related work, we describe the framework underlying our algorithm, building from abstract examples to the underlying theory and pseudocode. Finally, we provide a short evaluation of the algorithm as implemented in *Fiia.Net*.

2 Related Work

Model transformations are often applied to link two-level models of software architecture, as popularized by the OMG MDA [12] initiative. As software evolves, designers and developers make changes to both levels. Incremental bidirectional transformations allow these changes to be propagated through to the opposite model [13]. Unlike our runtime approach, however, these traditional techniques focus on static design and source code.

Most model transformations represent models as graphs. Triple Graph Grammars (TGGs) [14] are a common basis for incremental bidirectional transformations (e.g. [15–17]). TGGs are difficult to apply to graphs which have dissimilar structure [17], primarily because TGG rules map directly from source to target models without performing intermediate steps. This presents problems for our application to distributed systems: high-level behaviors are typically built from lower-level behaviors, and our rules naturally follow this structure using intermediate elements and non-determinism.

Other approaches for bidirectional model transformation impose a wide variety of restrictions [18]. QVT [19] defines two user languages: QVT Relational is similar to TGGs [20], and similarly does not support intermediate steps. QVT Operational, meanwhile, is an imperative language that would require hand-coded inverse rules to support bidirectionality. This relational versus operational split is typical of the remaining literature.

The Atlas Transformation Language (ATL) is a notable exception. It is an imperative model rewriting system, which has been extended to support conceptual adaptations and distribution removals [21]. The ATL-based technique is not stable with respect to local adaptations, so would cause unnecessary reconfiguration in live interactive systems.

Without supporting bidirectionality, Hearnden at al. [22] present a technique for incrementally updating a model transformation. Their formulation is based on maintaining the tree of possible derivations in a logic language.

Model transformations are also used in the area of distributed systems to specify adaptation between different configurations [23, 9]. These systems support high level specification of changes in a single-level model. We believe that similar techniques will be useful extensions to the *Fiia.Net* conceptual model API.

To the best of our knowledge, none of these systems satisfy our joint requirements of speed, generality, stability and limited bidirectionality.

3 Framework

The core of our approach is the use of two runtime models representing snapshots of the conceptual and distribution-level configuration of the distributed interactive system. These models are related via a *refinement* model transformation, while *adaptations* result in runtime changes to these models.

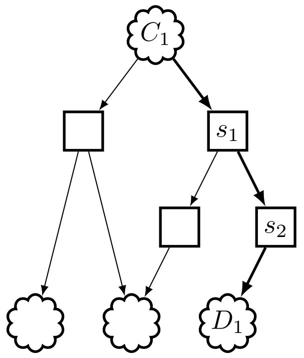


Fig. 1. One refinement in a tree of possibilities.

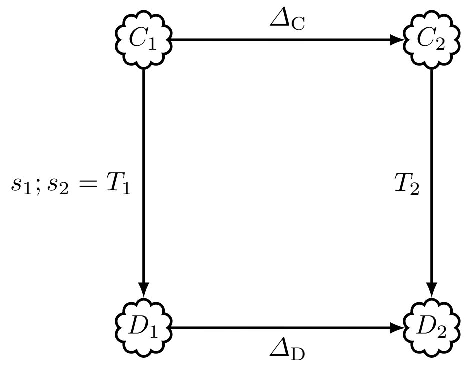


Fig. 2. Commuting of refinements over adaptations.

The high-level *conceptual* model is exposed to applications via reflection, events, and an editing API. Applications call into the API to specify runtime adaptations, and use events and reflection to gauge the effects of partial failure.

The low-level *distribution* model describes the underlying distributed system. It is modified automatically through refinement, as a result of conceptual adaptations. Partial failures are first reported as distribution adaptations, and then propagated up through the refinement to the developer's conceptual view.

Figure 1 shows the relationship between the models. A conceptual model C_1 corresponds to a distribution model D_1 through a sequence of transformation steps (such as s_1, s_2). Many different sequences of steps are possible, often leading to different distribution models. Figure 2 combines the steps into a single transformation T_1. The transformation steps are generated by non-deterministic application of a set of refinement rules.

At runtime, either model may be modified, as shown in Fig. 2. If a modification Δ_C is applied to C_1, the resulting conceptual model C_2 may not correspond with D_1. The runtime system must find a new transformation T_2 and distribution model D_2 which are compatible with C_2. Likewise, if a modification Δ_D is applied to D_1, the runtime system must find T_2 and a new conceptual model C_2.

The fact that updates are applied to both models excludes many transformation techniques. As we shall see in the following sections, limiting distribution adaptations to deletion allows our algorithm to support both general rules and fast, incremental updates. In summary, the main features of our approach are:

- Unordered graph rewriting rules are used to refine from conceptual to distribution models.
- Arbitrary changes may be applied to the conceptual model.
- Removals may be applied to the distribution model.
- Updates are applied stably, and at speeds suitable for interactive systems.

In the next sections, we expand on our algorithm and the underlying theory. We first describe how the model transformation is recorded into a trace of

```
$g \leftarrow$ the graph to refine;
**while** any rules match $g$ **do**
    $r \leftarrow$ any matching rule;
    **append** step $s$ to trace **where:**
        $s$.consumes = edges consumed by the rule;
        $s$.requires = edges required by the rule, but not modified;
        $s$.produces = edges produced by the rule;
    $g \leftarrow g - s$.consumes;                    /* Delete edges */
    $g \leftarrow g \cup s$.produces;                    /* Add edges */
**end**
the resulting distribution graph $\leftarrow g$;
```

Algorithm 1: **refine**: apply rewriting rules to a graph

steps. From there, we show how the trace and distribution are updated following changes in the conceptual model. Finally, we expand the algorithm to deal with removals from the distribution model.

4 Refinement and Trace

Refining the conceptual model into a distribution requires a very general refinement algorithm. Because there are multiple implementations of each conceptual model, the refinement must deal with non-determinism. Because the refinement involves a set of nested choices, the algorithm must permit the generation and consumption of *intermediate elements* to capture each refinement stage. Furthermore, adaptation must be permitted at either level. To the best of our knowledge, this combination is not addressed by existing techniques. Our algorithm solves a significant subset of this problem by supporting arbitrary adaptations at the conceptual level and removals at the distribution level. This section defines how we perform refinements and establish a trace. This information is then used to perform conceptual (Sect. 5) and distribution (Sect. 6) adaptations.

The relationship between conceptual and distribution models is specified via a set of graph rewriting rules. These rules are applied in arbitrary order until no more rules match, and their effects are recorded in a *trace*. This process is outlined in Alg. 1. As in other unordered graph rewrite systems, the rules can be very general: they need not be bidirectional, and can include non-determinism, intermediate elements, and multiplicities.

Rather than limit the behavior of individual rules, we enact adaptations by manipulating the trace. This section describes our formalization of the graphs, rules and trace.

The conceptual and distribution models are stored as graphs, each represented as a set of directed edges. Edges have a source vertex, a target vertex, and a label. Vertices are implicit, existing only as unique identifiers associated with an edge. All modifications to a graph can therefore be expressed via the removal and addition of edges.

Rewriting rules are represented as three sets of edges, plus subrules and executable code for advanced features. The three sets of edges describe most of the rule's behavior, so are where we focus this discussion. Each set describes a sub-graph, with the following meanings:

consumes: must exist for the rule to match, will be deleted;
requires: must exist for the rule to match, will not be modified;
produces: must not exist for the rule to match, will be added.

These sets describe both the prerequisites of a rule, and its effects when applied. This design is similar to many others, including the well-known double-pushout approach [24]. We now present an example of a *Fiia.Net* rule, and continue with the details of the trace.

In *Fiia.Net*, the conceptual model specifies a component-oriented distributed system. Components interact via explicit connectors, which express patterns of communication. For example, a *synchronization connector* between two components establishes them as copies of the same shared data, while a *stream* connector conveys realtime data such as sound or video. There are many possible ways to implement these connectors, expressed via a choice of refinement rules.

A much simpler connector is the *call connector*, which enables blocking method calls. In *Fiia.Net*'s rule set, a call connector can be implemented as a local pointer or remote procedure call (RPC). The remote procedure call can be direct, cached, or routed via a server.

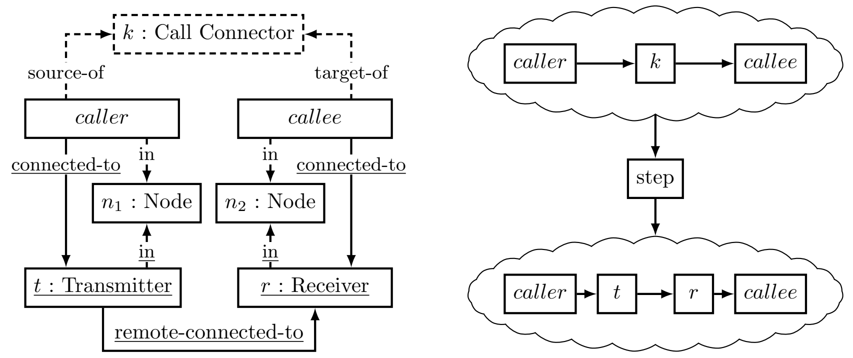
The *Fiia.Net* rule for rewriting a call connector into a direct RPC is sketched in Fig. 3(a). This rule deletes a call connector (k) between two endpoints (*caller* and *callee*) on different physical nodes (n_1 and n_2). It replaces the call connector with a network RPC link (t to r), as shown in Fig. 3(b). Call connectors appear in the *Fiia.Net* conceptual model, and are produced by many other rules. Even the *caller* and *callee* are often the product of earlier rules.

Matching a rule against a target graph consists of finding an *embedding.* An embedding is a mapping from a rule's precondition vertices (those consumed or required) to vertices in the target. As in other systems, the mapping must be injective, and applying it to the rule must produce a subgraph of the target. Vertices in the rule are either variables (e.g. k and *caller*), which could map to any one of many graph vertices, or exact values (e.g. "Call Connector").

If an embedding exists, the rule can be applied to the target graph. Rule application consists of deleting consumed edges and adding produced edges. Produced variables which do not appear in the embedding (e.g. t and r) are mapped to unique new vertices.

Each trace step records the effect and dependencies of a single rule application, i.e., the sets of edges consumed, required, and produced. The trace of a refinement is a sequence of trace steps recording all its rule applications.

A trace can sometimes be applied to graphs other than its original conceptual model. This process is shown in Alg. 2. If the graph is missing edges consumed or required by the trace, the **apply** will fail. In this case, the graph and trace are *incompatible.*



(a) Rule definition. Dashed elements are consumed, while underlined ones are produced.

(b) Rule application. This step replaces k with a network RPC link via t and r.

Fig. 3. Example *Fiia.Net* rule: Implement a synchronous call connector as an RPC link.

```
$t \leftarrow$ the trace to apply;
$g \leftarrow$ the original graph;
for each $s$ in $t$ do
    /* Ensure the preconditions are met.                              */
    if $s$.consumes $\not\subseteq g \vee s$.requires $\not\subseteq g \vee s$.produces $\cap\, g \neq \emptyset$ then
        raise the graph and trace are incompatible;
    end
    $g \leftarrow g - s$.consumes;                    /* Removed consumed edges. */
    $g \leftarrow g \cup s$.produces;                     /* Add produced edges. */
end
the resulting graph $\leftarrow g$;
```

Algorithm 2: **apply**: apply a trace to a graph

We handle negative and repeated patterns as subrules. If a negative pattern matches, it prevents the containing rule from matching. Repeated patterns match greedily, zero or more times, and contribute to steps produced from their containing rule.

The rules are applied in an unordered fashion. When multiple rules match, or multiple embeddings are possible, one is chosen arbitrarily[3]. The transformation continues applying rules until none match. This approach requires that the rule set be terminating and complete: all sequences of rule applications must be finite, and the final graph must be a distribution model.

[3] A steering algorithm can be attached to the rule refinery in order to guide these nondeterministic choices based on application-specific criteria, e.g. to minimize latency between components.

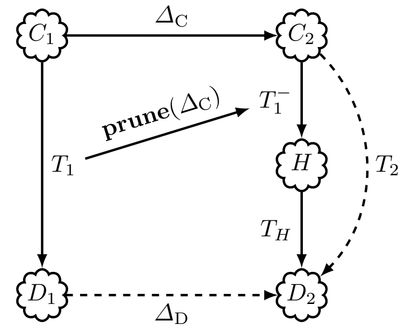


Fig. 4. Steps used in performing a conceptual adaptation.

```
$\Delta^+ \leftarrow$ edges added by the change;
$\Delta^- \leftarrow$ edges deleted by the change;
$g \leftarrow$ the graph to adapt;
$g \leftarrow g - \Delta^-$;     /* Delete edges */
$g \leftarrow g \cup \Delta^+$;        /* Add edges */
$g \rightarrow$ the resulting graph;
```

Algorithm 3: **adapt**: apply a change to a graph.

In the next sections, we build from these properties to the complete theory of our algorithm.

5 Conceptual Adaptations

Conceptual adaptations typically represent local changes within a larger model. Because the changes reconfigure a live system, they need to be propagated through the refinement quickly and incrementally. Existing techniques for such incremental updates greatly restrict the space of supported rules. Our algorithm solves this problem for unordered rewrite rules. This section describes how we apply conceptual adaptations to an existing refinement, using the explicit trace defined in Sec. 4.

Conceptual adaptations are defined using the graph representation of the conceptual model. An adaptation is a set of edges which are removed from the graph, and a set of edges which are added. Figure 4 shows the operations used to resolve a conceptual adaptation. From initial models C_1 and D_1, corresponding via trace T_1, and a conceptual adaptation Δ_C, our algorithm proceeds as follows:

1. Using Δ_C, **adapt** C_1 to C_2 (Alg. 3).
2. Using Δ_C, **prune** T_1 to a partial refinement of C_2, producing T_1^- (Alg. 4).
3. **apply** T_1^- to C_2, producing H (Alg. 2).
4. **refine** H, producing a trace T_H and model D_2 (Alg. 1).
5. Concatenate the steps of T_1^- and T_H, producing a new trace T_2.

The **prune** operation converts T_1 into a trace which represents a partial refinement of C_2. It does this by discarding steps which would not have been generated by a **refine** of C_2. Determining which steps to discard depends on the consumes, requires, and produces sets saved in each step. We now give an example of this pruning, and then present the complete algorithm.

Consider the abstract initial state shown in Fig. 5(a). Edges e_a and e_b exist in the conceptual model C_1. The first refinement step (s_1) consumes e_b and

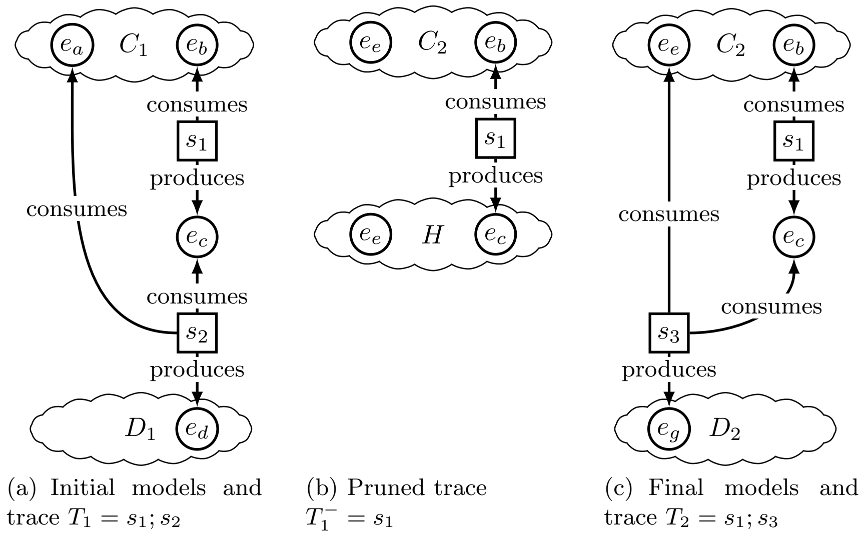


Fig. 5. Example conceptual adaptation $\Delta_C = \{$ **del:** e_a, **add:** e_e $\}$.

produces e_c. After the first step, the intermediate model consists of e_a and e_c. The second step (s_2) consumes both e_a and e_c, producing e_d. After the second step, no more rules match, so the distribution model consists of only e_d.

Now suppose that we apply a conceptual adaptation Δ_C that removes e_a and adds the new e_e. This causes the trace to be pruned to the T_1^- shown in Fig. 5(b). T_1^- is the same at T_1, except that steps that no longer apply in C_2 have been removed. Specifically, s_2 consumes e_a, so must be discarded. With the loss of s_2, e_d is no longer available. As neither e_c nor e_e are consumed in the trace, they appear in the intermediate model H.

Applying rewriting rules to H and concatenating the traces yields the T_2 shown in Fig. 5(c). s_1 remains unchanged from the initial state, but s_2 has been replaced by s_3.

The operational definition of **prune** is show in Alg. 4. It performs the dependency search outlined above based on the ordering of steps in the trace. Each step is defined from a graph rewrite operation in the transformation. As a result, all edges consumed or required by a step s_i in T_1 must appear in C_1, or be produced by a previous step (s_p where $p < i$). The iteration will always consider s_p before s_i, so can show dependency using a simple set intersection check.

For simplicity, this definition ignores negative and repeated patterns. In the complete algorithm, these are handled during the iteration. Negative patterns are checked against an intermediate graph when Alg. 4 considers the step compatible. Repeated patterns may expand or contract the step if their number of matches has changed. Both cases add to the complexity of the operation, but their use in *Fiia.Net* does not significantly impact runtime speed.

```
$t \leftarrow T_1$;
$r \leftarrow$ edges removed by $\varDelta_C$;
/* Propagate $\varDelta_C$ down through the trace.                    */
**for each** $s$ **in** $t$ **do**
    **if** removed $\cap$ $s$.consumes $\neq \emptyset$ **or** removed $\cap$ $s$.requires $\neq \emptyset$ **then**
        /* A prerequisite is unavailable, so delete this step.       */
        **delete** $s$ **from** $t$;
        removed $\leftarrow$ removed $\cup$ $s$.produces;
    **end**
**end**
$T_1^- \leftarrow t$;
```

Algorithm 4: **prune**: update the trace for a conceptual adaptation.

This technique allows us to quickly update an existing transformation with arbitrary conceptual changes. The resulting trace T_2 will always correspond to a possible sequence of rule applications on C_2, and so can be used in further adaptations. Unlike other approaches to live model transformation, our approach maintains general graph rewriting semantics throughout.

6 Distribution Adaptations

Partial failures in distributed systems are notoriously hard to resolve. A two-level runtime model provides a natural way of capturing this behavior: failures are removals in the distribution model, and are propagated back to the conceptual model. This allows the developer to work exclusively with the conceptual model, rather than delving into implementation details to diagnose and repair problems.

Our requirement for general rewriting rules excludes existing techniques for bidirectional adaptation. Restricting distribution adaptations to removals allows us to apply them quickly and incrementally. We are not aware of any other algorithm for unordered rewrite rules that offers this capability. This section defines how we handle distribution adaptations, building on operations defined earlier.

Like conceptual adaptations, distribution adaptations are specified as graph edits; however, they are restricted to removals. Figure 6 shows the operations used to resolve a distribution-level adaptation. Initial models C_1 and D_1 correspond via trace T_1. The distribution adaptation $\varDelta_D$ produces a new distribution D_i from D_1. The unrestricted rule set implies that D_i might not correspond to any conceptual model. Our algorithm resolves this conflict by removing additional distribution edges to restore consistency ($\varDelta_{\mathrm{D2}}$). This whole operation is performed as follows:

1. **findSourceDelta** with $\varDelta_D$ and T_1 to generate $\varDelta_C$ (Alg. 5).
2. Using $\varDelta_C$, **adapt** C_1 to C_2 (Alg. 3).
3. Using $\varDelta_C$, **prune** T_1 to its parts compatible with C_2, producing T_1^- (Alg. 4).

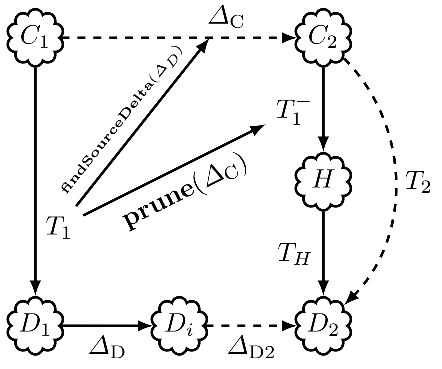

C_1
$\mathit{\Delta}_{\mathrm{C}}$
C_2
findSourceDelta$(\mathit{\Delta}_{\mathrm{D}})$
T_1^-
H
T_2
prune$(\mathit{\Delta}_{\mathrm{C}})$
T_1
T_H
D_1
D_i
D_2
$\mathit{\Delta}_{\mathrm{D}}$
$\mathit{\Delta}_{\mathrm{D2}}$

Fig. 6. Steps used in performing a distribution adaptation.

4. **apply** T_1^- to C_2, producing H (Alg. 2).
5. **refine** H, producing a trace T_H and model D_2 (Alg. 1).
6. Concatenate the steps of T_1^- and T_H, producing a new trace T_2.

The **findSourceDelta** operation finds conceptual removals $\mathit{\Delta}_C$ sufficient to cause $\mathit{\Delta}_D$. The derived $\mathit{\Delta}_C$ is then applied like a normal conceptual update, following the algorithm described in Sect. 5. This conceptual update often removes more distribution elements than $\mathit{\Delta}_D$, causing the additional $\mathit{\Delta}_{D2}$.

Consider the abstract example shown in Fig. 7(a). We apply a distribution update $\mathit{\Delta}_D$ which removes e_d. The task of **findSourceDelta** is then to identify the conceptual edges which led to e_d, so they can be removed. e_d was produced by s_2, which, in turn, consumes e_a. As a result, $\mathit{\Delta}_C$ will remove e_a.

Applying **prune** with $\mathit{\Delta}_C = \{$ **del:** e_a $\}$ produces the T_1^- trace shown in Fig. 7(b). Without e_a, pruning discards both s_2 and s_3. This leaves only e_c, no longer consumed by s_3, to appear in the intermediate model H.

To refine H to a valid distribution model, we rely on the earlier property of completeness. The trace T_1^- describes a sequence of rule applications from the conceptual model C_2 to H. In our example, e_c happens to be an intermediate element, which can not appear in the distribution model. For the completeness property to hold, applying the refinement rules must eventually refine H into a distribution model. In this example, s_4 is sufficient, and yields the final T_2 shown in Fig. 7(c).

The operational definition of **findSourceDelta** is shown in Alg. 5. It performs the inference described above by walking backward through the trace. As for **prune**, all edges consumed by a step must appear in C_1 or be produced by a previous step. This iteration always considers the consume before the produce, so can accumulate the banned edges in "removed".

This technique quickly updates an existing transformation to apply distribution removals. Our approach is particularly unique, because it does not require bidirectional rules. Indeed, the transformation used in our *Fiia.Net* system is neither bijective nor surjective, and is massively non-deterministic.

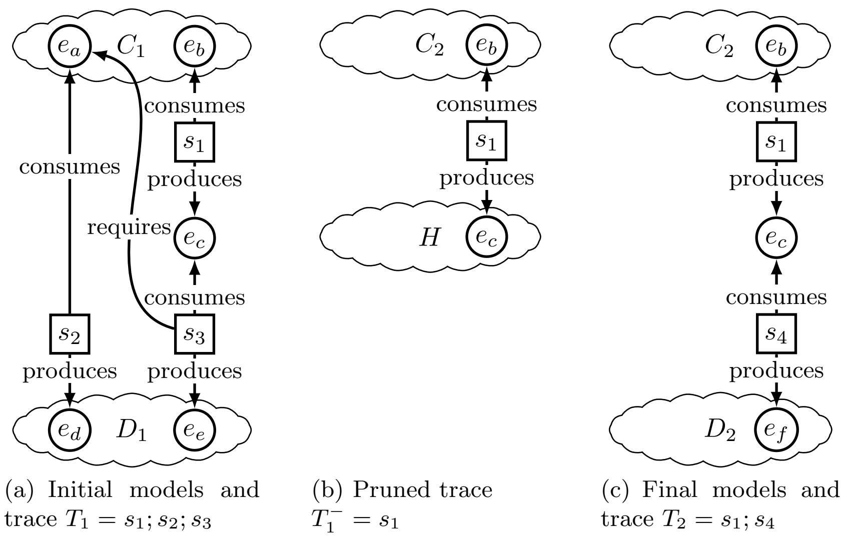


Fig. 7. Example distribution adaptation $\Delta_D = \{ \textbf{del: } e_d \}$.

7 Experience

The algorithm presented in this paper is used by our *Fiia.Net* toolkit. *Fiia.Net* represents a distributed interactive system using a high-level conceptual model which is visible to the application, and a low-level distribution model which configures the actual implementation [10]. An application enacts adaptations by modifying the conceptual model, while the underlying implementation removes any failed elements from the distribution model. Our algorithm efficiently propagates both types of updates through the transformation. *Fiia.Net*'s rule set consists of 34 graph rewrite rules, each of which is simple, but which in combination express a rich set of possible implementations for each conceptual model.

We have used *Fiia.Net* to implement several applications within our lab. These include a shared presentation program; a multimodal furniture layout [10] involving participants using an electronic tabletop surface, a standard PC, and a smartphone; a textual chat application; and a collaborative game prototyping tool [11]. These examples have shown the effectiveness of the two-level model for groupware, and the practicality of using *Fiia.Net* toolkit for rapid application development.

To evaluate the performance impact of our model transformation algorithm, we have recorded the time it requires for various adaptations based on the Raptor game prototyping tool [11]. Raptor allows designers to add and control in-game entities while a tester plays. Each in-game entity is a single *Fiia.Net* component. Adding, removing, and connecting entities causes Raptor to make changes in the

```
$T \leftarrow T_1$;
$\Delta^- \leftarrow$ edges removed by $\Delta_D$;
/* Propagate $\Delta_D$ up to the conceptual model.                    */
for each $s$ in reverse $T$ do
    /* Check whether this step is compatible with the change.       */
    if $\Delta^- \cap s$.produces $\neq \emptyset$ or $\Delta^- \cap s$.requires $\neq \emptyset$ then
        delete $s$ from $T$;          /* Discard the incompatible step. */
        /* Discarding this step means more edges should not exist.   */
        $\Delta^- \leftarrow \Delta^- \cup s$.consumes;
    end
end
$\Delta_C \leftarrow$ { del: $\Delta^- \cap \Delta_C$ } ;      /* Compute the conceptual adaptation. */
```
Algorithm 5: **findSourceDelta**: find a sufficient conceptual removal for a distribution removal

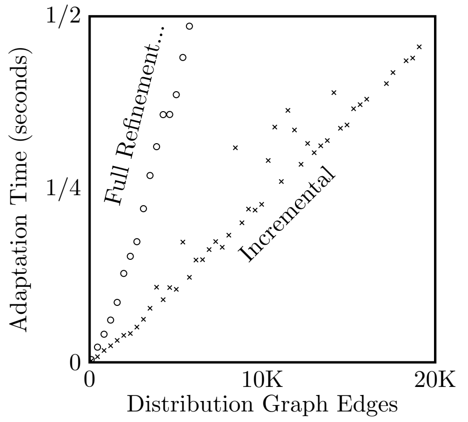
conceptual model. Similarly, partial failures will cause changes to the distribution model.

Figure 8(a) compares the incremental algorithm, described in this paper, to a straight-forward full refinement. In both cases, the experiment gradually introduces 1000 new entities into a game world via conceptual adaptations. All transformation is performed in one thread, on an Acer Aspire 5110 (AMD TL-50 1.6GHz with 2GB RAM, running Windows XP and Microsoft .Net 3.5). The size of the trace and models grows linearly with the number of entities. At 1000 entities, it reaches 8022 steps, with 16054 and 19076 edges in the conceptual and distribution graphs, respectively. Applying a full refinement after each adaptation rapidly becomes too expensive for interactive applications, peaking at nearly 4 seconds. Our incremental algorithm performs much better, remaining below 500 ms. This gap appears because **prune** preserves almost all of the previous steps, so the ensuing **refine** requires few graph searches.

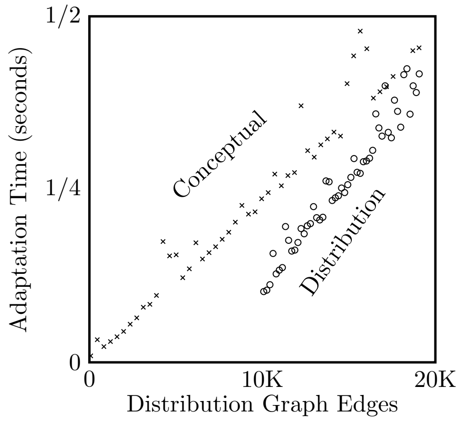
Figure 8(b) shows the same system, removing entities from the game world via conceptual and distribution adaptations. Applying the adaptations in the conceptual model behaves similarly to the incremental additions. The distribution adaptations are slightly different, because removing components from the distribution model indicates their failure. To ease application recovery, *Fiia.Net* preserves some information about connections to failed components. This behavior is responsible for both the slightly higher performance of, and the 10076 distribution edges remaining after the distribution case. Again, performance is adequate for interactive use.

Our current graph rewriting engine is relatively crude. It stores all intermediate models as untyped graphs, and dynamically matches rules using recursive search. In spite of these shortcuts, our current implementation works well with a few thousand components and connections.

While our approach is motivated by the difficulties of developing distributed systems, the algorithm is independent of *Fiia.Net*'s models and rules.



(a) Conceptual adaptations to add game entities.



(b) Conceptual and distribution adaptations to remove game entities.

Fig. 8. Adaptation times for varying model sizes.

8 Conclusion

In this paper, we have presented an efficient algorithm for maintaining consistency in two-level runtime models. This allows systems like *Fiia.Net* to maintain all the flexibility of model-driven architecture, in a highly-adaptive and fault-tolerant runtime.

Because our algorithm is built on graph rewriting and tracing, it should also permit many optimizations and heuristics that we have not explored. We believe that this approach will prove useful for similar two-level runtime models, whether specialized for groupware or other fields.

References

1. Graham, T.C.N., Kazman, R., Walmsley, C.: Agility and experimentation: Practical techniques for resolving architectural tradeoffs. In: ICSE, IEEE Computer Society (2007) 519–528
2. Pinelle, D., Dyck, J., Gutwin, C.: Aligning work practices and mobile technologies: Groupware design for loosely coupled mobile groups. In Chittaro, L., ed.: Mobile HCI. Volume 2795 of Lecture Notes in Computer Science., Springer (2003) 177–192
3. Achterbosch, L., Pierce, R., Simmons, G.: Massively multiplayer online role-playing games: the past, present, and future. Computers in Entertainment **5**(4) (2007)
4. Graham, T., Urnes, T.: Linguistic support for the evolutionary design of software architectures. In: ICSE 18, IEEE Computer Society (1996) 418–427
5. Calvary, G., Coutaz, J., Nigay, L.: From single-user architectural design to PAC*: A generic software architecture model for CSCW. In: CHI '97, ACM Press (1997) 242–249

6. Hill, R., Brinck, T., Rohall, S., Patterson, J., Wilner, W.: The *Rendezvous* language and architecture for constructing multi-user applications. ACM TOCHI **1**(2) (June 1994) 81–125
7. Laurillau, Y., Nigay, L.: Clover architecture for groupware. In: CSCW '02, ACM Press (2002) 236–245
8. Graham, T., Phillips, W., Wolfe, C.: Quality analysis of distribution architectures for synchronous groupware. In: CollaborateCom. (2006)
9. Morin, B., Fleurey, F., Bencomo, N., Jézéquel, J.M., Solberg, A., Dehlen, V., Blair, G.S.: An aspect-oriented and model-driven approach for managing dynamic variability. In: MoDELS. (2008) 782–796
10. Wolfe, C., Graham, T.N., Phillips, W.G., Roy, B.: Fiia: User-centered development of adaptive groupware systems. In: EICS (to appear). (2009)
11. Wolfe, C., Smith, J.D., Phillips, W.G., Graham, T.N.: Fiia: A Model-Based Approach to Engineering Collaborative Augmented Reality. In: The Engineering of Mixed Reality Systems (to appear). Springer (2009)
12. OMG: MDA guide version 1.0.1. Technical Report omg/03-06-01, OMG (2003)
13. Kurtev, I.: State of the art of QVT: A model transformation language standard. In Schürr, A., Nagl, M., Zündorf, A., eds.: AGTIVE. Volume 5088 of Lecture Notes in Computer Science., Springer (2007) 377–393
14. Schürr, A.: Specification of graph translators with triple graph grammars. In Mayr, E.W., Schmidt, G., Tinhofer, G., eds.: WG. Volume 903 of Lecture Notes in Computer Science., Springer (1994) 151–163
15. Becker, S.M., Westfechtel, B.: Incremental integration tools for chemical engineering: An industrial application of triple graph grammars. In Bodlaender, H.L., ed.: WG. Volume 2880 of Lecture Notes in Computer Science., Springer (2003) 46–57
16. Giese, H., Wagner, R.: Incremental model synchronization with triple graph grammars. In Nierstrasz, O., Whittle, J., Harel, D., Reggio, G., eds.: MoDELS. Volume 4199 of Lecture Notes in Computer Science., Springer (2006) 543–557
17. Kindler, E., Wagner, R.: Triple graph grammars: Concepts, extensions, implementations, and application scenarios. Technical Report tr-ri-07-284, Department of Computer Science, University of Paderborn (June 2007)
18. Stevens, P.: A landscape of bidirectional model transformations. In Lämmel, R., Visser, J., Saraiva, J., eds.: GTTSE. Volume 5235 of Lecture Notes in Computer Science., Springer (2007) 408–424
19. OMG: Meta object facility (MOF) 2.0 query/view/transformation specification. Technical Report formal/2008-04-03, OMG (2008)
20. Greenyer, J., Kindler, E.: Reconciling TGGs with QVT. In: MoDELS. (2007) 16–30
21. Xiong, Y., Liu, D., Hu, Z., Zhao, H., Takeichi, M., Mei, H.: Towards automatic model synchronization from model transformations. In Stirewalt, R.E.K., Egyed, A., Fischer, B., eds.: ASE, ACM (2007) 164–173
22. Hearnden, D., Lawley, M., Raymond, K.: Incremental model transformation for the evolution of model-driven systems. In: MoDELS. (2006) 321–335
23. Bencomo, N., Blair, G.S., France, R.B.: Summary of the workshop models@run.time at models 2006. In: MoDELS Workshops. (2006) 227–231
24. Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R., Löwe, M.: Algebraic approaches to graph transformation - Part I: Basic concepts and double pushout approach. In Rozenberg, G., ed.: Handbook of Graph Grammars, World Scientific (1997) 163–246