Chapter 6
Conclusion and Future Work

The works presented as part of this book are novel and yet limited in terms of their applicability to real-world enterprises. There is ample scope for further exploration of the problems identified as part of this research. The algorithms and results presented in this book can be further modified and adapted for better application to enterprises. This chapter is divided into two sections. Section 6.1 presents a brief summary of our research contributions in this book. Section 6.2 presents some exciting future research directions that can be explored to benefit the GORE community.

6.1 Summary of the Work

In this book, we present a collection of novel solutions that aims to improve the state-of-the-art as far as enterprise modelling and requirements analysis is considered in goal-oriented requirements engineering. We address an enterprise modelling scenario that had not been considered by the community previously. We highlight the importance of goal modelling in enterprise hierarchies and particularly underline the importance of an ontology integration framework for such goal model hierarchies (also referred to as requirement refinement hierarchies). We present one such framework for integrating the ontologies between adjacent level goal models and measuring the degree of correlation that exists between them. We also establish the fact that goal model hierarchies are not merely a hypothetical concept and they really manifest themselves in real world event logs. The relevance of this research stems from the fact that we could mine adjacent and non-adjacent hierarchic structures from real-world data.

Apart from enterprise modelling, we have also contributed to the GORE community by enhancing the existing state-of-the-art in terms of requirements analysis. We have identified that very limited research had been done to perform model checking on goal models. We have proposed a new heuristic called the *Semantic Implosion Algorithm* and simulated it to compare its performance with an existing heuristic that

N. Deb and N. Chaki, *Business Standard Compliance and Requirements Validation Using Goal Models*, Services and Business Process Reengineering,
https://doi.org/10.1007/978-981-15-2501-8_6

was proposed by Fuxman [1]. The new heuristic has been shown to outperform the existing heuristic by a factor of almost 10^{17}. Thus, our proposed solution is much more efficient and scalable when it comes to performing model checks. We have also implemented our proposed algorithm by developing a tool, called i*ToNuSMV. The tool accepts goal models in the textual GRL (tGRL) notation and temporal properties in CTL. The underlying model checker that has been integrated into our tool is NuSMV. Analysts can now feed goal models into the i*ToNuSMV tool and check them against temporal compliance rules. NuSMV generates counter-examples whenever a CTL property is not satisfied by the goal model.

Finally, we underline the importance of going beyond the structural features and orderings, and analysing the semantics of goal model configurations. The AFSR framework proposed therein enables the modeller to annotate goals with their associated semantics. The framework has a semantic reconciliation machinery that can evaluate the semantics of any goal as obtained from its subgoals. These derived and intended semantics can then be compared to perform semantic analyses like entailment and consistency checks. AFSR does not stop here; it goes beyond conflict detection by suggesting re-factored solutions that are conflict-free. This framework can be deployed for real-world goal model maintenance and their adoption in evolving requirement settings. However, exploring the entire space of goal model configurations for identifying the optimal solution manually seems to be quite impractical and erroneous. Human effort often leads to suboptimal solutions. We have shown how this situation can be tackled by mapping the goal model maintenance problem to the state space search problem. Establishing the admissibility and consistency of our heuristic path cost function has allowed us to deploy A* search over the space of goal model configurations, thereby, guaranteeing the optimal solution.

6.2 Future Research Directions

In this section, we try to shed some light on the future research directions emanating from the works presented in this book. A greater insight into the impact of enterprise hierarchies on goal modelling techniques can be derived from the work on goal model hierarchies. We observe from the data mining exercise on real-world data that employees within an organization need not necessarily follow the structure of the hierarchy. We have mined non-adjacent hierarchic correlations from the data as well. The proposed framework for requirement refinement hierarchies works with adjacent level hierarchies only. This framework can be extended to non-adjacent levels as well, thereby, developing a system to measure the correlation of the entire requirement refinement hierarchy. The works on requirements analysis can be extended as discussed in the following sections.

6.2.1 Extracting Business Compliant Finite State Models

The i*ToNuSMV tool is evolving quite rapidly. Version 2.02 of the i*ToNuSMV tool supports multi-actor goal models having inter-actor dependencies. It also supports model checking with CTL constraints. However, we have been working on a major release that will derive constrained finite state machines from a goal model. This implies that instead of feeding a goal model as input and then checking a temporal constraint on the derived FSM, we will provide the constraint along with the goal model as input and the derived FSM will already satisfy the given constraint.

6.2.1.1 Assumptions

The different types of CTL constraints have been studied in detail and this paper works with a finite subset of such constraints in the framework. The primary goal is to generate a compliant finite state model by pruning transitions from the finite state model generated by i*ToNuSMV ver2.02. The proposed guidelines have the following four assumptions:

A-1 Since FSMs are derived for fulfilment of goals, the framework works with only `AG` and `EG` temporal operators for the violation of goal fulfilment. Example:`AG(V109!=FU)`.

A-2 Two CTL predicates can be connected through Boolean connectives like AND and OR. This framework allows the user to define only two predicates at a time and connect them by the`AND` or `OR` operator. Example: `AG(V109!=FU AND V102!=FU)`.

A-3 Another type of CTL constraint that is addressed is implication (→). Any two constraint can have implication between them. The implication operator has been restricted to only single level of nesting. Example: `AG(V101=CNF` → `AF(V102=FU AND V103!=FU))`.

A-4 The goal tree level for an actor has been assumed to be 3 to reduce the problem complexity.

6.2.1.2 CTL Properties Handled

This section briefly explains each of the CTL constraints that were addressed in [2] and how the corresponding finite state models are derived.

1. **EG(V#! = FU) for AND-decomposition**. These types of properties are safety properties that prevent something bad from happening. Ensuring this property on a goal with AND-decomposition requires the pruning of `CNF` → `FU` transitions for some subset of the child nodes.

2. **EG(V#! = FU) for OR − decomposition**. The same type of safety property on a goal with OR-decomposition has different consequences. Ensuring such a property requires the pruning of `CNF` → `FU` transitions for all the child nodes.
3. **EG(V#! = FU AND V#! = FU)**. CTL properties which have multiple CTL predicates connected with boolean AND connectives can be ensured by satisfying each predicate separately. The solution space is a Cartesian product of models generated from each CTL predicate—denoted by $M \times N$.
4. **EG(V#! = FU OR V#! = FU)**. CTL properties having multiple CTL predicates connected with boolean OR connectives can be ensured by satisfying either of the predicates or both. The solution space is much larger and denoted by $M + N + (M \times N)$.
5. **AG(V#! = FU → V#! = FU)**. These type of CTL properties (defined with the implication operator →) specify an ordering over the fulfilment of goals. Thus, all those invalid states need to be pruned from the FSM that violate this property. State transitions to or from these invalid states are correspondingly removed.
6. **EG(V#! = FU) for AND-OR-decompositions**. Ensuring such safety properties for multilevel goal models with OR-decompositions nested under an AND-decomposition requires the pruning of `CNF` → `FU` transitions for all OR-children. This needs to be done for any subset of the AND-children of the root node.
7. **EG(V#! = FU) for OR-AND-decomposition**. If the root goal is OR-decomposed followed by each OR-child undergoing an AND-decomposition, then these types of CTL properties can be ensured by pruning `CNF` → `FU` transitions for any subset of AND-children for each of the OR-child of the root goal.

The above seven types of CTL properties have been addressed in the newly proposed version 3.0 of the $i^*ToNuSMV$ framework. For a more detailed understanding of how each of these CTL property classes is ensured within a goal model, readers can refer to [2]. Figure 6.1 demonstrates the workflow of the $i^*ToNuSMV$ 3.0 framework.

6.2.1.3 Demonstration with Case Study

In this section, the working of the $i^*ToNuSMV$ 3.0 deployment interface is demonstrated with the help of a simple real-life case study. Figure 6.2 shows a simple goal model that captures the requirements for `Access Locker`. It requires two tasks to be performed—`VerifyCodeTrue` verifies whether the user access code entered is true and `GiveAccess` finally gives the access of the locker to the user provided the code entered is true. An intuitive CTL property associated with this goal model is also shown in the figure. `AG(V103!=FU → V104!=FU)` implies that the task `GiveAccess` cannot be performed until the task `VerifyCodeTrue` is successfully completed.

The $i^*ToNuSMV$ 2.02 tool, which implements the Semantic Implosion Algorithm (SIA), generates a finite state model irrespective of the CTL property associ-

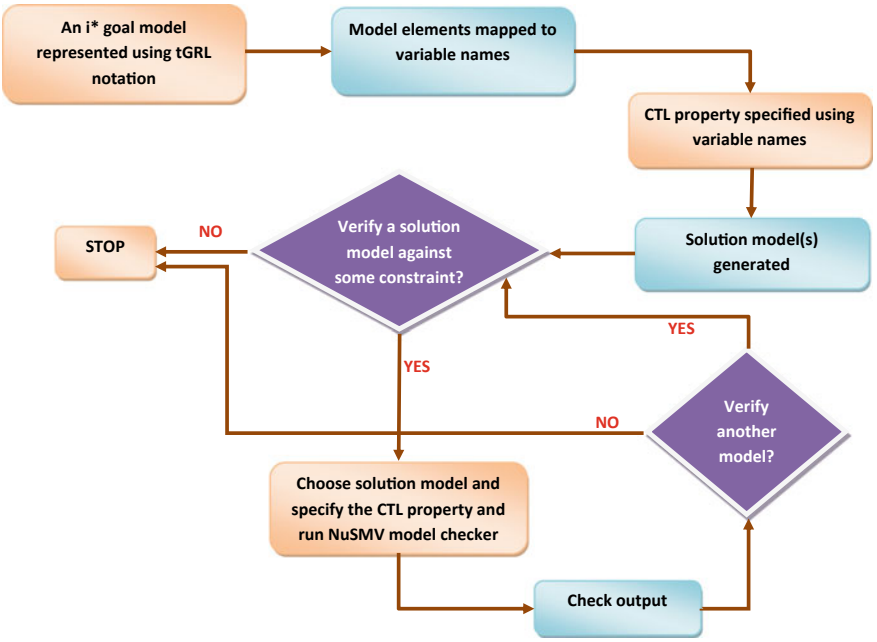
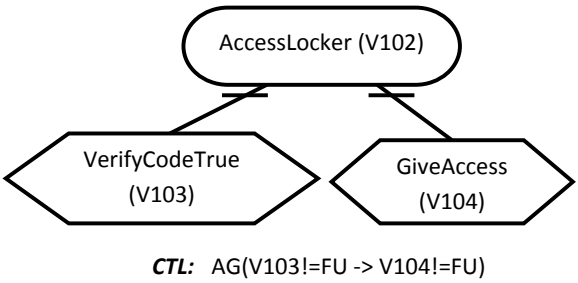


Fig. 6.1 The workflow of the $i^*ToNuSMV$3.0 deployment framework

Fig. 6.2 A simple goal model for accessing a locker



ated with the goal model. Since the goal model in Fig. 6.2 has a two child AND-decomposition, the corresponding FSM has a 2-dimensional lattice structure for capturing all possible execution sequences to fulfil the root goal. The derived FSM is shown in Fig. 6.3.

The research guidelines proposed in [2] have been implemented in *i*ToNuSMV 3.0*. It is an extension of the Semantic Implosion Algorithm that takes the finite state model generated by SIA and prunes those transitions which violate the given CTL property. The pruned finite state model for the given goal model and CTL property (refer to Fig. 6.2) is shown in Fig. 6.4.

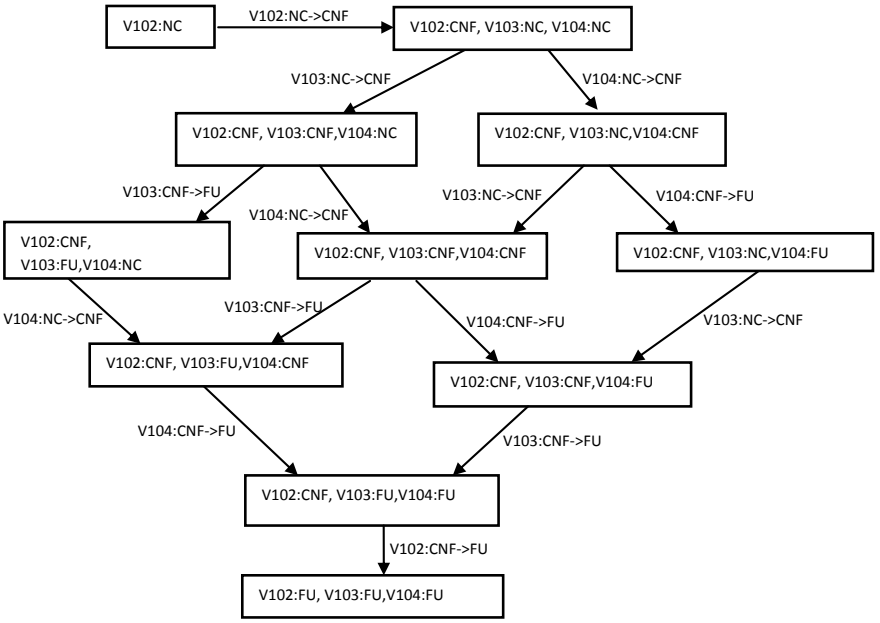


Fig. 6.3 FSM generated by *i*ToNuSMV 2.02*

6.2.1.4 URL

The *i*ToNuSMV 3.0* framework can be downloaded from the following link: https://github.com/istarToNuSMV/i-ToNuSMV3.0. The User Manual and use case examples have been shown on the webpage.

6.2.1.5 Experimental Results

In this section, some experimental results have been documented that were obtained after performing extensive simulations with the existing (version 2.02) and newly proposed (version 3.0) versions of the $i^*ToNuSMV$ framework. Arbitrary goal models were designed with varying complexity in terms of the number of actors, the number of goals, the number of AND/OR-decompositions and the complexity of the associated CTL constraints. The simulations did not bring out any anomalous behaviour. Data were collected with respect to the number of transitions in the final output FSM and the execution time.

The bar chart of Fig. 6.5 shows a comparative analysis between SIA (implemented in version 2.02) and Complaint-SIA (implemented in version 3.0). *i*ToNuSMV 2.02* does not generate a compliant FSM like *i*ToNuSMV 3.0*. Thus, the FSM generated by version 2.02 includes all possible execution sequences between sets of states. The complaint-FSM generated by version 3.0 will have fewer number of transitions as

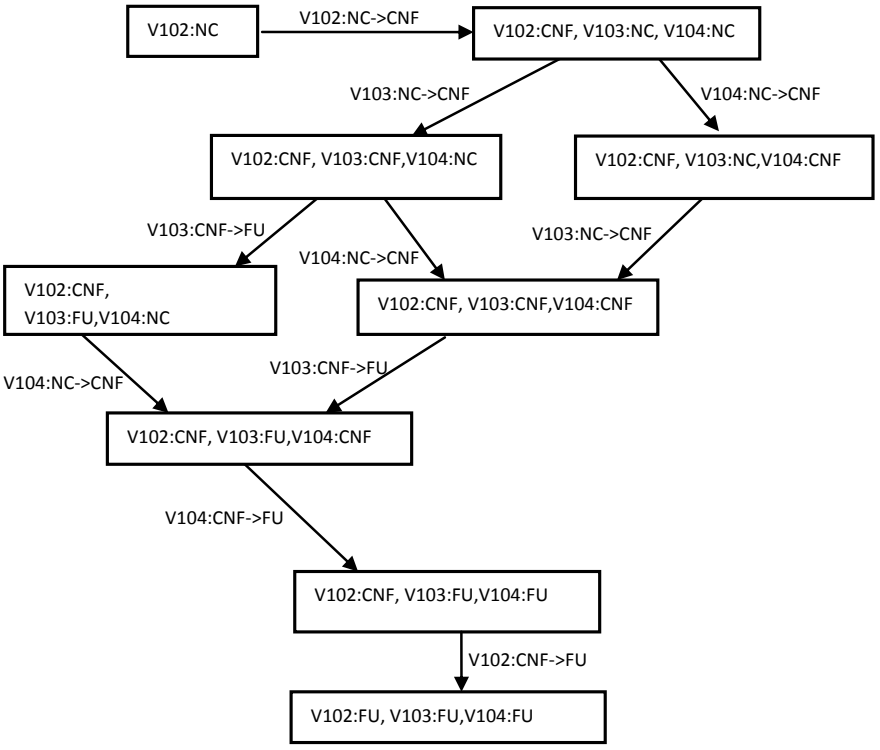


Fig. 6.4 FSM generated by *i*ToNuSMV 3.0*

all CTL properties used in these simulations, impose some sort of ordering between events. This results in the final FSM having only a subset of the transitions included by SIA. The degree (or %) of reduction in state space is dependent on several factors rather than only one.

The line plot shown in Fig. 6.6 compares the execution time of SIA and Complaint-SIA—both measured in milliseconds. With the same set of simulation parameters, it is observed that *i*ToNuSMV 3.0* takes much more time than *i*ToNuSMV 2.02* to generate the finite state models. This is also quite logical as version 3.0 implements some additional checks and tasks after SIA is executed (as in version 2.02). Basically, version 3.0 takes the FSM generated by SIA and individually scans and prunes transitions to satisfy the given CTL property. Also, as discussed in [2], there may be multiple strategies for pruning different subsets of transitions in order to satisfy the CTL property. *i*ToNuSMV 3.0* executes each such strategy and generates a unique finite state model (pruned and compliant) for each of these strategies. This is the reason why version 3.0 takes much longer to reach completion.

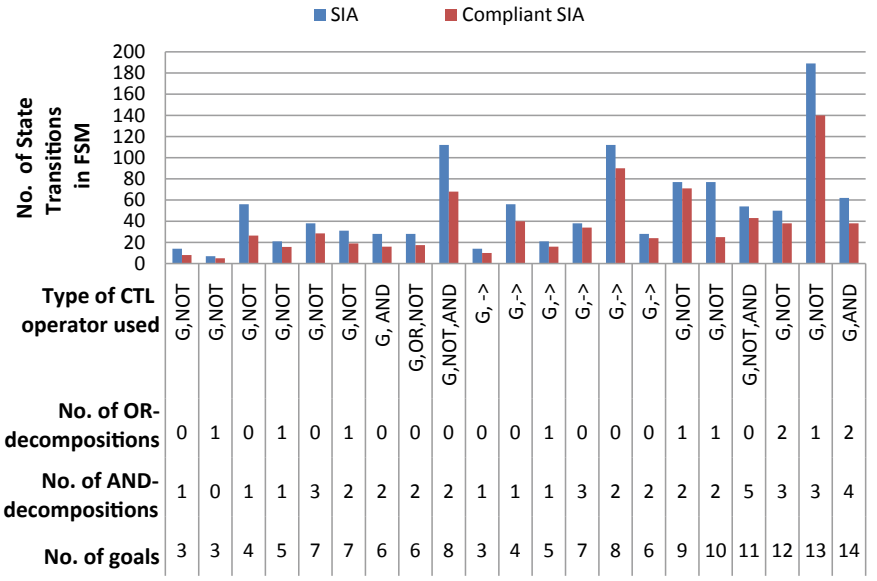


Fig. 6.5 Number of state transitions in the final FSM

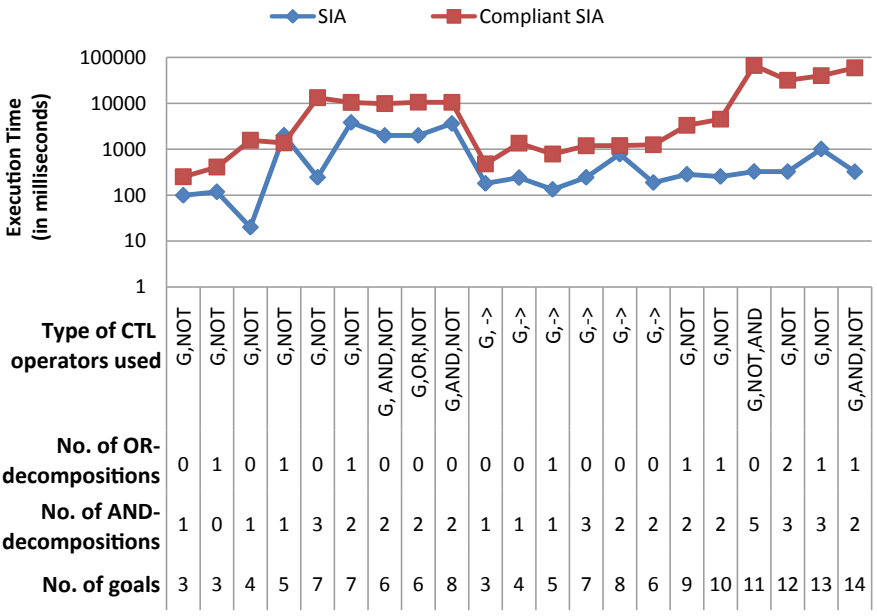


Fig. 6.6 Execution time for deriving the final FSMs

6.2.1.6 Conclusion and Future Work

In [2] the authors only presented some use cases to demonstrate how CTL compliance can be ensured in finite state models. They also documented an algorithm for the process. However, a proper deployment framework implementing the guidelines was missing. This paper builds on the guidelines proposed in [2] and presents a proper deployment interface for i*ToNuSMV 3.0. It provides the URL for downloading and installing the framework and the different features supported by the interface (with the help of a case study). It also measures and compares the performance of the newly built version with the existing version of the i*ToNuSMV tool (also see Table 6.1).

This work presents a tool to generate finite state models from goal models that already satisfy some given CTL constraint. Finite state models can be more readily transformed into code. Thus, this research takes an important step towards the development of business compliant applications directly from goal models. Most business compliance rules have some sort of temporal ordering over events and can be represented with temporal logics efficiently. However, the proposed solution has several assumptions which needs to be relaxed for making the framework more complete.

One of the more important limitations of the proposed solution is that only one temporal property (in CTL) can be specified along with the goal model specification. Future versions of the $i^*ToNuSMV$ framework will aim to allow users to specify multiple CTL properties over a single goal model specification. Another limitation of the new version is the extra processing time that is required. The additional pruning mechanism requires extensive checking of the finite state model generated by SIA. Currently, research efforts are being channelized to develop an efficient version of the Semantic Implosion Algorithm that will generate compliant FSMs in a more efficient manner.

Table 6.1 Feature comparison between verions 2.02 and 3.0

Features	*i**ToNuSMV 2.02	*i**ToNuSMV 3.0
Input specification	*i**goal model defined using tGRL	*i**goal model defined using tGRL and a CTL property
Number of FSM generated	1	1 or more than one
Compliant FSM	FSM may or may not be compliant to any temporal property	FSM compliant with a given CTL property
Solution space	Comparatively large	Reduced solution space
Number of NuSMV input	1	One for each of the FSM generated
Verification	NuSMV model checker verifies property on single finite state model	NuSMV model checker can separately verify each of the solution models

6.2.2 The CARGo Tool

The AFSR framework has been presented along with an implementation roadmap that uses A* search. This research direction has several avenues that can be further explored to make it more applicable to enterprises. For instance, we have worked with functional semantic annotations only. Research can be directed to incorporate non-functional semantics associated with softgoals. Non-functional semantic analyses can enrich the mechanism for choosing between multiple strategies of goal satisfaction. Also, the most imminent research scope is to build a proper tool interface that implements the AFSR framework.

The *CARGo* prototype [3] is built on the AFSR framework. It makes the use of a data structure, as illustrated in the following section, for representing and modifying goal models. Algorithm 1 shows the main procedure of the prototype and how this data structure is used.

6.2.2.1 Semantically Annotated i* Networks (SAi* Nets)

SAi* Nets are a non-linear data structure representation of goal models that have been developed for the *CARGo* tool prototype. It is similar to an adjacency list (a list of linked lists) where each list captures the strategic rational model of a specific actor. Every list is headed by an *actor_node* which specifies the particular actor. Each goal model element within the actor's goal tree is represented using *tree_nodes* that have the node structure shown in Fig. 6.7. A sample abstracted SAi* Net representation is shown in Fig. 6.8. All computations and modifications proposed by the AFSR framework, for identification and removal of annotation conflicts, is implemented on the SAi* Net. The final conflict-free SAi* Nets are translated to textual goal model descriptions for end-user readability. Each *tree_node* has the following fields:

- *val*: An integer used to identify each goal model element uniquely.
- *str*: Name of the element.
- *type*: Integer values are used to identify decomposition type of the *tree_node*—0 for OR-decompositions, 1 for AND-decompositions and 2 for leaf nodes.

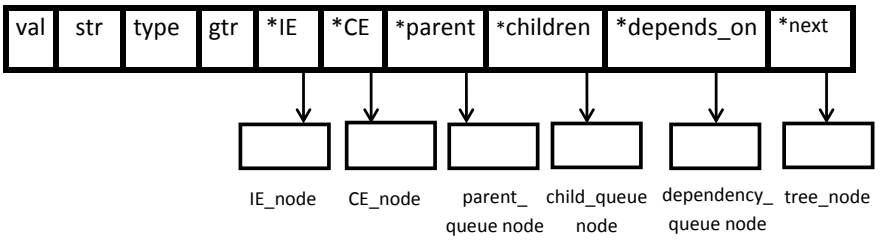


Fig. 6.7 *tree_node* structure

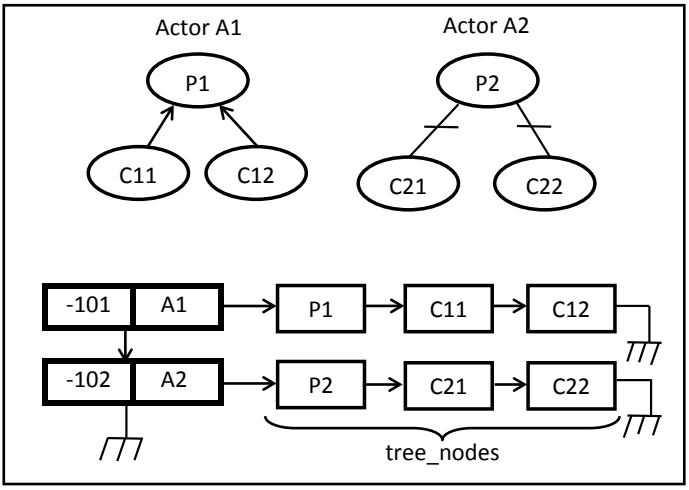


Fig. 6.8 Abstracted view of an example SAi* Net

- *gtr*: Integers used to identify goal, task or resource.
- **IE*: List of immediate annotations as first-order logic predicates.
- **CE*: List of cumulative annotations as evaluated by SRA.
- **parent*: A pointer to its parent queue. For root nodes, this field is NULL.
- **children*: A pointer to its child list. For leaf nodes, this list is empty.
- **depends_on*: A pointer to the list of dependencies associated with that *tree_node*. For an independent node, this list is empty.
- **next*: A pointer to the next *tree_node* in the actor's goal model.

6.2.2.2 Platform Used

The back end code is generated in the C language. The front end design is developed in JAVA.

6.2.2.3 URL of the CARGo Prototype

The *CARGo* tool can be freely downloaded from the following URL: https://github.com/CARGoTool/CARGoV1.0.

Algorithm 1 CARGo_tool

Input: i* model with immediate annotations in textual format.
Output: Conflict-free i* model variant in textual format.
Data Structure: SAi* network and a *conflict_list*.

```
1: procedure MAIN
2:     Build SAi* network from given input i* model
3:     Compute CE of each node by traversing SAi* network
       in bottom-up approach
4:     Traverse SAi* network either in top-down or in
       bottom-up approach according to given user choice
       and generate the conflict_list.
5:     do
6:         Extract node from the conflict_list.
7:         Perform ERA or CRA as per the type of conflict
8:         Apply SRA to update the SAi* network and con-
           flict list.
9:     while conflict_list is not empty
10:    Generate i* model representation of the conflict free
       SAi* network in textual format
11: end procedure
```

6.2.2.4 Benefits of the CARGo Tool

The annotation of goal model artefacts within a goal model is not one of the major contributions of this tool. It is somewhat similar to the annotation mechanism supported by jUCMNav for GRL. In fact, we work with tGRL goal models. The main benefit of the tool is in the domain of goal model maintenance in changing business environments. The tool helps with the adaptation of goal models when business requirements change. Changing requirements cause a change in the relative contexts of the goal model artefacts. The *CARGo* tool identifies the conflicts arising out of these changes in contexts. Conflict resolution is performed by refactoring the goal model and creating a goal model variant that is conflict-free. Existing goal modelling analysis techniques can be applied to all goal model variants as well. Thus, the *CARGo* tool helps in the evolution of goal models in changing business environments.

6.2.2.5 Conclusion

The *CARGo* tool is *sound* as the output is always a conflict-free goal model. It is also partially *complete* with the exception of softgoals and softgoal contexts. The number of iterations for conflict resolution is nondeterministic as it depends on the number and type of conflicts observed in the initial goal model.

6.2.3 *Building Mobile Applications from Goal Model Specifications*

This section elaborates the generalized framework of the GRL2APK tool. Figure 6.9 illustrates the framework and Sect. 6.2.3.1 explains the role of the individual components. The framework is generic and does not depend on any specific technology; it is, however, limited to only structural NFRs. The following sections elaborate on the workflow of the GRL2APK tool and a real-life use case illustrating how the tool can be used to generate Android applications.

6.2.3.1 Architecture of the New Framework

The overall architecture of the GRL2APK framework is depicted in Fig. 6.9. In this framework, we propose an integration of different components and services that allows enterprise architects to build applications while having the flexibility to choose the operationalizations of the specified NFRs. This framework has four underlying assumptions:

1. The code modules of the functional requirements are stored in a cloud repository.
2. The desired NFRs have to be specified by the enterprise architects within the goal model description. System developers only need to choose the "*operationalization*" of the desired NFRs.
3. NFR catalogs for high-level NFRs have been developed by specialized requirement engineers and stored in a cloud repository for reuse.
4. The functional codes for implementing the operationalizations of *structural* NFRs have also been stored in the code repository.

The proposed GRL2APK framework has the following components:

- **Goal Model Specifier**: The first module provides an interface to the enterprise architects to design the goal model based on the end-user requirements. Enterprise architects can use any goal requirements language to capture such models. We suggest the `Extended tGRL` (or, XtGRL) language.
- **Component Extractor**: The goal model is then passed through this module to go through the `XtGRL` grammar artefacts as specified by the enterprise architects. The module scans through the input model and extracts all the high-level NFRs and functional goal decompositions that have been specified.
- **NFR Fetcher**: After extraction of the specified high-level NFRs, the *NFR Fetcher* module is invoked. This module fetches the NFR catalogs of all those high-level NFRs that have been identified in the previous phase. It is based on assumption (3). It is a 1-to-1 mapping that allows this module to fetch the required NFR catalogs.
- **NFR Repository**: This repository stores two types of information—the *NFR catalogs* for high-level NFRs and a *NFR conflict database* that identifies conflicts across NFRs and their operationalizations. Such a repository can be stored on a local server or in some cloud repository like Google Firebase or Amazon AWS. NFR

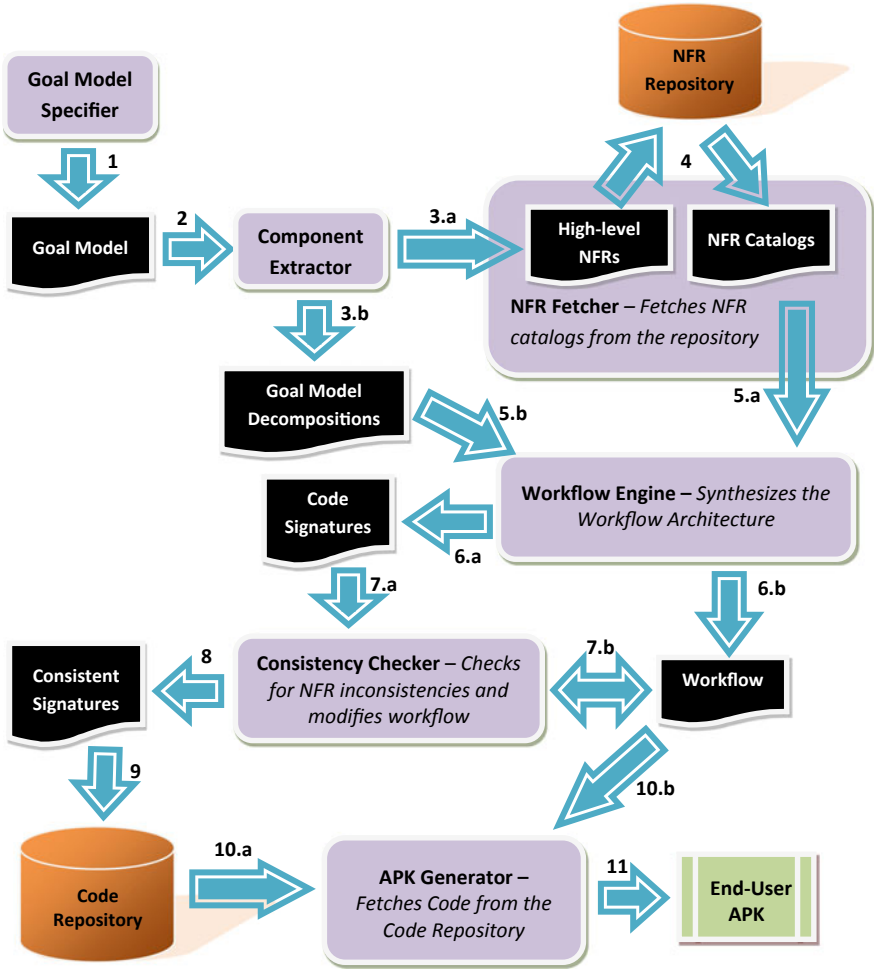


Fig. 6.9 The proposed framework for app orchestration from goal models using selective composition of NFRs

catalogs are static in nature as they only capture the decomposition of high-level NFRs into low-level operationalizations. The NFR conflict database is dynamic and needs to be updated based on available NFR operationalizations and also on the particular application vertical where the framework is being deployed.

- **Workflow Engine**: The goal decompositions (from Step-2) and the NFR Catalogs (from Step-3) are fed into the *Workflow Engine* that provides an interface where the system developer has to choose between different operationalizations and code signatures for both functional and non-functional requirements.
- **Consistency Checker**: The developer may choose operationalization strategies that conflict with other high-level NFRs. This module alerts the developer of the

existence of such conflicts. The developer, however, has the choice to prioritize a particular operationalization, thereby, ascertaining the satisfaction of the NFR. The Workflow file is correspondingly updated and the consistent set of Code Signatures are also identified.

- **APK Generator**: This is the final phase of the framework that takes the consistent set of Code Signatures and the Workflow to generate an APK file for end-users. By the time the framework reaches this phase, all operationalizations of specified high-level NFRs have been decided and all conflicts (if any) have been resolved with the help of developer prioritization.
- **Code Repository**: This repository stores two types of codes—*Functional Requirement (FR) Codes* and *NFR Codes*. FR Codes are used to implement specific functionalities represented by goals and tasks. NFR codes are used to capture operationalizations of structural high-level NFRs.

The FR codes within the Code repository may be developed by software developers (who may or may not specialize in Requirements Engineering). The NFR Repository and the NFR codes are typically created, updated and managed by requirement engineers who are well-trained and experts in NFR management. The Code repository may be built incrementally—the greater the availability of FR and NFR code components, the richer is the quality of the App generated by the GRL2APK framework. The GRL2APK approach is aimed at driving towards the automation of app generation based on the availability of integrable code components within a code repository. However, validation of the app being generated with respect to the requirements captured in the original goal model still remains a necessity.

The GRL2APK tool is built on the newly proposed framework with the help of mainly four technologies: *Acceleo* (a platform for code generation), *Google Firebase* (cloud storage for NFR catalogue repository), *Amazon AWS S3* (cloud storage for functional code repository) and *Java Services* (used at the back end for consistency checking and app generation). We will elaborate on each of these technologies and how they have been used for building the GRL2APK tool (Fig. 6.10).

6.2.3.2 Components of the GRL2APK Tool

The GRL2APK tool provides a guideline (only) as to how the different components of the GRL2APK framework (shown in Fig. 6.9) can be realized using state-of-the-art technologies. System developers can choose among alternate available technologies for realizing any of the components.

- **Acceleo** [4]: We provide as input a goal model written in `XtGRL` to the Acceleo platform of the Eclipse tool. Acceleo is an Eclipse-based product created and developed by the Eclipse Strategic Member Obeo. Acceleo uses Model to Text language (MTL) to extract the component of a model. It supports Java services behind the scene to process these kinds of domain-specific languages. Acceleo extracts the

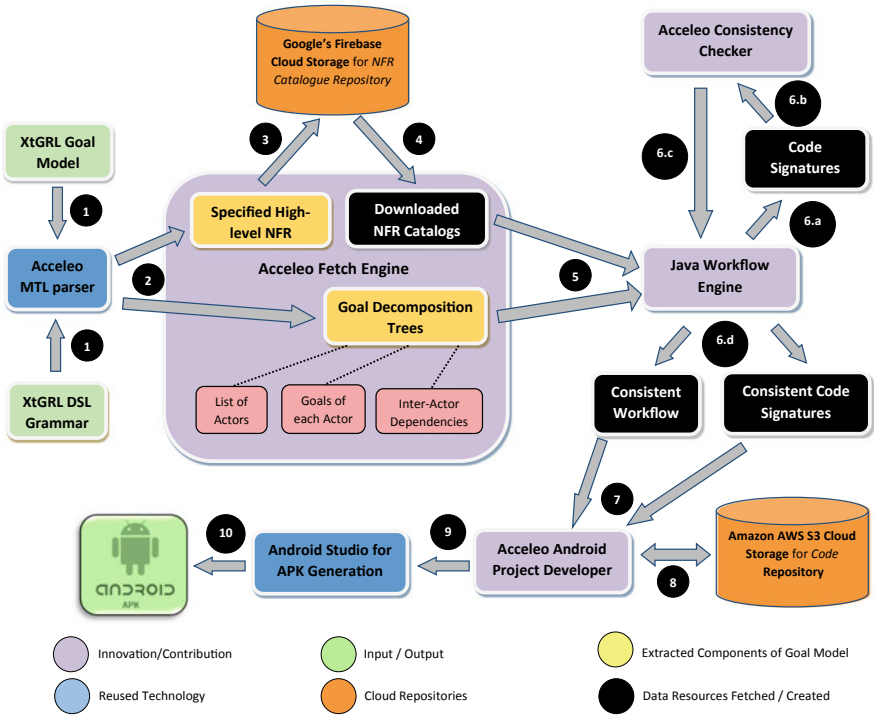


Fig. 6.10 The implementation framework with the process steps numbered in black circles

necessary information from the input requirements model. It also gives the provision to write Java services to accomplish specific tasks.

`Acceleo Modules`: The *Component Extractor* and *NFR Fetcher* modules in Fig. 6.9 are implemented using this technology. We call it the `Acceleo MTL Parser` and `Acceleo Fetch Engine`, respectively, in Fig. 6.10.

- **Google Firebase** [5, 6]: One of the vital assumptions of the proposed framework is to access NFR catalogs from a cloud repository as per the specified high-level NFRs. We use Google Firebase cloud service where we can store any kind of files. A dozen of Google Firebase cloud storage APIs provide flexibility to access specific NFR catalogs as specified in the requirements model. Another important aspect of the Google Firebase cloud storage is that we can authenticate every user of the application with ease.

  `Firebase Module`: The *NFR Fetcher* (shown in Fig. 6.9) uses Google Firebase APIs to download NFR catalogs from the NFR Repository. This component is called `Acceleo Fetch Engine` in Fig. 6.10.

- **Amazon AWS S3** [7]: AWS S3 is one of the leading object storage cloud services that allows accessing, storing and analysing any amount of data securely from anywhere. We have chosen S3 as the functional code repository. Amazon claims the durability of AWS S3 is about 99.99%. Several APIs are available to access functional codes stored in AWS S3 using Java services in Acceleo platform and integrate them with the other modules. Alternatively, *Google Cloud*, *Microsoft AZURE* or other cloud services could also be used for these repositories.

  `AWS Module`: The *APK Generator* (Fig. 6.9) uses AWS S3 APIs to integrate the functional code corresponding to the given goal model and chosen NFR operationalizations. We call it the `Acceleo Android Project Developer` in Fig. 6.10.

- **Java Services**: Finally we use several Java services to generate Android source codes from extracted components of the requirements model and NFR components of the NFR catalogs. We also integrate necessary files and dependencies of Android libraries and generate .APK file (Android Application Package) with Java services.

  `Java Modules`: The *Consistency Checker, Workflow Engine* and *APK Generator* heavily use Java Services. The corresponding modules in Fig. 6.10 are called `Acceleo Consistency Checker, Java Workflow Engine` and `Acceleo Android Project Developer`, respectively.

6.2.3.3 Workflow of the GRL2APK Tool

Input: Goal model specification capturing functional and high-level non-functional requirements, a cloud repository for NFR catalogs and another cloud repository storing functional code.

Output: An Android .APK file implementing the operationalizations of structural NFRs as selected by the developer.

Process Steps: (see Fig. 6.10)

*Step*1 Goal model specification and the `XtGRL` CFG is fed into the `AcceleoMTL Parser`.

*Step*2 MTL modules and Java services in the `Acceleo MTL Parser` process the goal model and extract the necessary components—high-level NFRs and goal decomposition trees.

*Step*3 According to the "demands"of the softgoals within the goal model, specific NFR catalogs (stored in the Google Firebase cloud storage) are accessed by the `Acceleo Fetch Engine`.

*Step*4 Google Firebase APIs are used to download those catalogs.

*Step*5 NFR Catalogs and Goal Decompositions are fed into the `Java Workflow Engine`.

*Step*6 The framework iteratively derives a set of Code Signatures that are conflict-free as follows:

(a) The developer selects his desired Code Signatures for FRs and NFR operationalizations.
(b) The NFR operationalizations are checked for conflicts in the `Acceleo Consistency Checker`.
(c) In case of an NFR conflict, the developer is prompted to choose another set of operationalizations. The corresponding Code Signatures are collected and the process is repeated.
(d) If there is no conflict, then the `Java Workflow Engine` generates the Consistent Workflow and Code Signatures for generating the APK.

*Step*7 The Consistent Workflow and Code Signatures are fed into the `Acceleo Android Project Developer`.

*Step*8 The `Acceleo Android Project Developer` uses Amazon AWS S3 APIs to access the code repository and download the actual code components.

*Step*9 The `Acceleo Android Project Developer` creates the Android Studio project for APK generation while making necessary changes to the root *AndroidManifest.xml* file.

*Step*10 The Android project created in `Step-9` is fed into Android Studio for compilation and building of the APK file.

6.2.3.4 Generating a Remote Healthcare Android App

We have considered remote healthcare system for our case study. It refers to the ongoing healthcare project "*A Framework for Healthcare Services using Mobile and Sensor cloud Technologies*" under the Information Technology Research Academy *ITRA*.[1] The project coordinators agreed to share their code repositories that would help in the generation of Android applications using our proposed framework. We modelled a part of the system consisting of some functional goals and some NFRs like `Security` and `Data-space Performance`. In this section, we present in detail how the framework is executed in a real-life scenario. The necessary screenshots for every phase has been provided for better visualization.

The XtGRL Goal Model

In this section, we create a scenario where an actor Patient wants to submit his medical details. The corresponding goal *ProvideMedicalDetails* "demands" the high-level NFRs *Security* and *Data-space Performance*. The input XtGRL goal model is as follows:

[1] Project URL: https://itra.medialabasia.in/?p=632.

```
grl Health Care{
      actor Patient{
            goal SeekHealth care{
                  decompositionType ='and';
                  decomposedBy ProvideMedicalDetails,
                  SendReports,GetMedicine;
            }
            goal ProvideMedicalDetails{
                  demands Security;
                  demands Data-space Performance;
            }
            softGoal Security;
            softGoal Data-space Performance;
      }}
```

NFR Catalogs

The more important functional requirement in the goal model, with respect to the proposed framework, is the `ProvideMedicalDetails` goal. This goal "demands" two different high-level NFRs—`Security` and `Data-space Performance`. The NFR catalogs corresponding to these two high-level NFRs are as follows:

```
nfrl catalog{
      nfr__SGoal Security{
            decompositionType=or;
            decomposedBy AES_Encryption, DES_Encryption;
      }
      op_SGoal AES_Encryption;
      op_SGoal DES_Encryption;
}
nfrl catalog{
      nfr_SGoal Data-space Performance{
            decompositionType=or;
            decomposedBy PPM, LZ, CM;
      }
      op_SGoal PPM;
      op_SGoal LZ;
      op_SGoal CM;
}
```

Workflow Engine Interface

Once the NFR catalogs are downloaded, the Workflow Engine allows the developer to decide the control flow between different goals and tasks as well as the implementation code signatures for both functional and non-functional requirements.

Consistency Checker

Once the Code Signatures are selected by the developer, the Consistency Checker module checks the chosen signatures against a NFR conflict database (that is stored in the NFR repository). In case of conflicts, the module shows a prompt as seen in Fig. 6.11. The value of 30% is derived from the conflict database. If there are no conflicts between the chosen operationalizations (for example, in our case study, Lempel Ziv (LZ) for Data-space Performance and AES_Encryption for Security), then the Workflow Engine creates the workflow file as shown in Fig. 6.12.

APK Generator

The APK Generator can now download the code modules based on the code signatures that are mentioned in the workflow file (Fig. 6.12). The workflow file captures the order of execution of goals (or tasks), which in this case study turns out to be provideMedicalDetails(), followed by send_reports() and get_medicine(). The work-

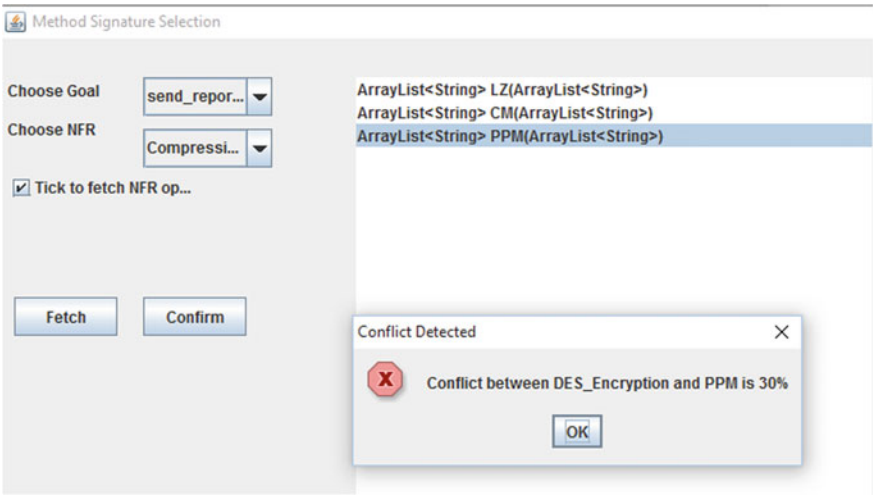


Fig. 6.11 Conflict identified between operationalization PPM (for Data-space performance) and DES_Encryption (for security)

```
void provideMedicalDetails(ArrayList<String>)
>> ArrayList<String> LZ(ArrayList<String>)
>> ArrayList<String> AES_Encryption(ArrayList<String>)
>> ArrayList<String> send_reports()
>> void get_medicine(ArrayList<String>)
>> void SeekHealthcare()
>> stop()
```

Fig. 6.12 Workflow generated for LZ (for Data-space performance) and AES_Encryption (for security)

flow also captures the order in which the NFR operationalizations have to be applied. The provideMedicalDetails() module passes the patient data (accepted as argument) to the LZ() code module for compression. The compressed data is then passed to AES_Encryption() module for encrypting before storage.

Figure 6.13 shows a screenshot of the app that is generated with the help of Android Studio for the above case study. Figure 6.14 shows how the data are stored in the Patient database. For illustration purposes, we included two dummy operationalizations—No_Encryption() and No_Compression()—to show the proper functioning of the GRL2APK framework based on developer's choice. A careful inspection of Fig. 6.14 shows a medical record of patient "ajit pal" which is being submitted from the app interface shown in Fig. 6.13. The selection of No_Encryption() and No_Compression() by the developer resulted in storing this data in the database as-is. Another patient data on the lower side of Fig. 6.14 shows how the data has been stored after applying LZ() compression followed by AES_Encryption(). Thus, depending on the developer's choice of NFR operationalizations, the generated app behaves differently.

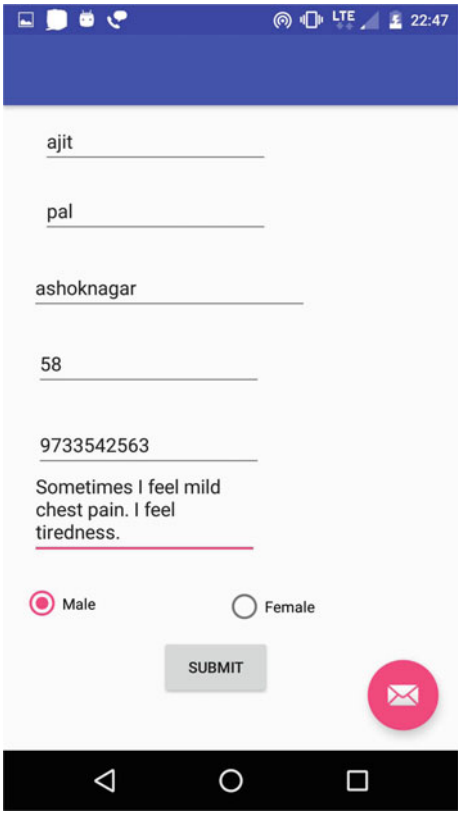


Fig. 6.13 Screenshot of the app

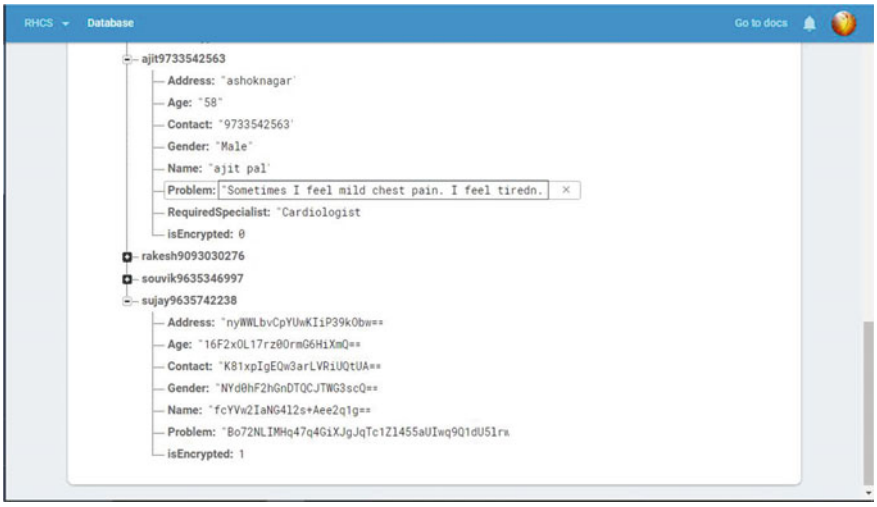


Fig. 6.14 Screenshot of patient database

References

1. Fuxman AD (2001) Formal analysis of early requirements specifications. MS thesis, Department of Computer Science. University of Toronto, Canada
2. Deb N, Chaki N, Roy M, Bhaumik A., Pal S Extracting business compliant finite state models from i* models. In: Advanced computing and systems for security (ACSS), advances in intelligent systems and computing, vol. 995. Springer, Singapore. ISBN: 978-981-13-8962-7
3. Deb N, Mallik M, Roychowdhury A, Chaki N Cargo: a prototype for contextual annotation and reconciliation of goal models. In: Accepted in the 27th international IEEE requirements engineering conference (RE)
4. Musset J, Juliot É, Lacrampe S, Piers W, Brun C, Goubet L, Lussaud Y, Allilaire F (2006) Acceleo user guide, vol. 2
5. Moroney L (2017) Moroney, Anglin, definitive guide to firebase. Springer. https://doi.org/10.1007/978-1-4842-2943-9
6. Stonehem B (2016) Google android firebase: learning the basics, vol. 1. First Rank Publishing
7. AWS (2016) Amazon simple storage service developer's guide. https://s3.cn-north-1.amazonaws.com.cn/aws-dam-prod/china/pdf/s3-dg.pdf