Chapter 4
Model Checking with i*

Hierarchic correlation between adjacent levels of a goal model hierarchy only ensures the synchronization between different levels of the enterprise hierarchy. It does not ensure the compliance of goal models to business compliance rules. Model checkers or verifiers can do this type of analysis. Model checking is a method for formally verifying finite state concurrent systems represented by extended finite state models. Industry standard model checking tools—like SPIN [1], NuSMV [2]—accept these extended finite state models (E-FSM) as input. The input models are defined by a set of state transitions that characterize the possible execution traces that the system can generate. The model checking tools are also fed with specifications about the system, expressed using temporal logic. Efficient symbolic algorithms are used to traverse the model defined by the system and check if the specification holds or not. Thus, either a positive acknowledgement is generated, if the specification is satisfied, or a counterexample is produced, if the specification is violated. Figure 4.1 illustrates the general working mechanism of model checkers.

Requirement models capture the requirement specifications of the system and have the same impact as design models have on the coding phase [3]. Requirement models are generated in the requirements analysis phase of software development. This is the first phase of developing a software or system, irrespective of the particular development life cycle model being followed—*Waterfall, Spiral, Prototype* or *Agile*. Requirement models can help enterprise architects and developers by allowing them to perform different kinds of analysis on the system being developed. Model checking against a given set of temporal properties (see Fig. 4.1) is also an important type of analysis that may be performed on goal-oriented requirement models.

Even after a system has been deployed, its environment keeps changing. This results in the user requirements to change as well. The system has to adapt to the ever-changing user needs during runtime. Before incorporating the newly evolved requirements into the existing framework, developers and architects need to ensure that the changing requirements do not result in conflicting/inconsistent states within the system. Thus, some kind of model checking needs to be done on the changed requirements to ensure that the system will remain consistent after the changes take

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_4

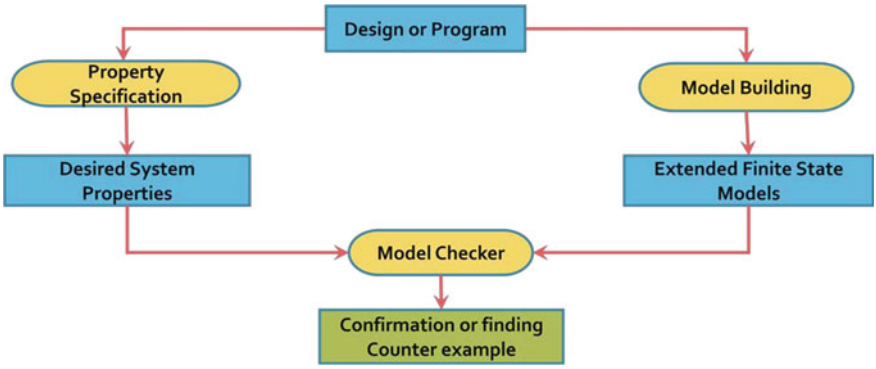

Fig. 4.1 Block diagram of standard model verifiers

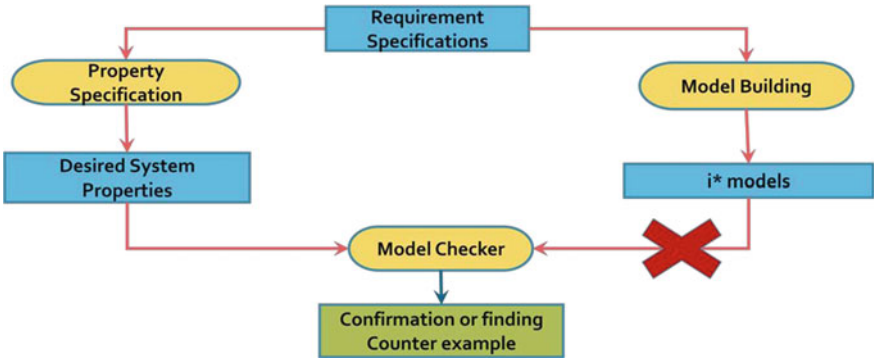

Fig. 4.2 Problem with i* model verification

effect. This may be done with industrial model checkers, if they are fed with the updated requirement models and the evolved requirement specifications as shown in Fig. 4.1.

i* is a goal-oriented requirements modelling notation that models requirements with the help of actors and their goals, tasks and resources. Inter-actor dependencies are also captured by the i* framework. An inherent attribute of the i* notation (and goal models in general) is that it is sequence agnostic and does not capture any sort of partial ordering between the goals and tasks. In the absence of sequencing information, standard industrial model checkers cannot verify i* models against temporal property specifications as shown in Fig. 4.2. Model checkers accept extended finite state models (E-FSM) as input for verifying temporal properties. Process models, sequence diagrams or activity diagrams, capture some ordering of states within the system and, hence, E-FSM(s) can be easily derived from these models. The process of extracting E-FSM(s) from i* models is far more complex as i* models do not capture state transitions within the system. This is the underlying research question being addressed in this chapter and is the main motivation behind this research.

Formal Tropos introduces the concept of actor instances and how dependencies, assertions, possibilities and invariants can exist in either of three states—*Not Created*, *Created Not Fulfilled* and *Fulfilled* [4, 5]. Formal Tropos associates the Tropos methodology to a formal specification language that allows the specification of constraints, invariants, pre- and post-conditions, thereby capturing the semantics of the i* graphical models. We extend this notion to i* model constructs in general and state that every goal, task or resource also exists in either of these three states. Every model construct makes two state transitions to reach the *Fulfilled* state from the *Not Created* state. The *Naïve Algorithm* uses a brute-force method to generate all possible finite state models that can be obtained by permuting the state transitions of individual model constructs. This results in an explosion within the finite state model space.

It is interesting to observe that, although an i* model is sequence agnostic, yet there exists some features or model constructs within the i* model that provide a temporal insight into the underlying requirements of the enterprise. For instance, every dependency has a *cause-effect property* in the sense that it is only when a dependee satisfies a requirement of the depender does the dependency become fulfilled. The *Semantic Implosion Algorithm* identifies these untapped temporal characteristics and tries to contain the rate of growth of the finite state model space corresponding to an i* model. Simulation results reveal that the *Semantic Implosion Algorithm* indeed outperforms the *Naïve algorithm* and provides a drastic improvement over the brute-force method.

The rest of the chapter is structured as follows. The next section (Sect. 4.1) details out the *Naïve Algorithm* and the *Semantic Implosion Algorithm*. The drawbacks of the *Naïve Algorithm* are identified and the *Semantic Implosion Algorithm* is proposed as a solution to these drawbacks. Section 4.2 describes a detailed simulation where both the algorithms are applied to the same classes of i* models and their performances are observed, compared and contrasted. Section 4.3 presents the `i*ToNuSMV` tool that we have developed for implementing the *Semantic Implosion Algorithm* and performing model checks on i* models. concludes the paper. This is followed by the version history and web links of the tool in Sects. 4.4 and 4.5, respectively. Finally, we conclude the chapter in Sect. 4.6.

4.1 Developing Finite State Models from an i* Model

The main research motivation is to analyse an i* model and derive a finite state model (FSM) that captures all the finite execution sequences that satisfy the given i* model. Without identifying a partial ordering of the operations within the enterprise, it becomes very difficult to check and verify temporal properties and compliance rules on the system. The underlying challenge of this work lies in the fact that i* models are sequence agnostic. Being complementary to the notion of FSMs, which define an ordering of states through which the system can go through, the conversion process cannot yield a unique execution trace corresponding to a given i* model. The idea here is to generate all the possible execution traces that satisfy the requirement

specifications captured in the given i* model. The algorithms presented in this article produce a finite state model space, as output, which defines this set of plausible finite state sequences. Once the finite state model space is obtained, we can apply model checking and generate a subset of this model space which satisfies all the compliance rules necessary for the operation of the enterprise. This final set of pruned finite state sequences can then be reverted back to the enterprise owner in order to verify the requirements.

In the following algorithms we are considering the more detailed strategic rationale (SR) diagram of an i* model. The SR-diagram is much more comprehensive than its strategic dependency (SD) counterpart and encompasses all the dependency information that are captured in the SD-diagram. In fact, an SD-diagram only represents the inter-actor dependencies but does not depict which particular model construct of the depender is dependent on which particular model construct of the dependee. The SR model is much more elaborate in this sense.

4.1.1 The Naïve Algorithm

The most intuitive solution that has been given by the GORE community uses the notion that every i* model element can exist in either of three possible states—*Not Created (NC), Created Not Fulfilled (CNF),* and *Fulfilled (F)*. All the goals, tasks and resources that appear in the SR model are initially in the *Not Created (NC)* state and every model construct must make two transitions to reach the *Fulfilled (F)* state while transitioning through the intermediate *Created Not Fulfilled (CNF)* state. Unlike Formal Tropos [5], we do not consider multiple instances of goals, tasks or resources. We assume single instances and derive a finite state model corresponding to the given i* model. We obtain sequences of states by considering all possible permutations of the model elements and the states in which they exist.

Let us demonstrate the above concept with an example. Consider the simplest possible SR-diagram with one actor consisting of only one goal G. This is shown in Fig. 4.3a. The goal G can be in either of three states—*Not Created* denoted by $G_{(N)}$, *Created Not Fulfilled* denoted by $G_{(C)}$ and *Fulfilled* denoted by $G_{(F)}$. These three states give rise to 3! finite state sequences as shown in Fig. 4.3b–g. However, out all these six finite state models, only Fig. 4.3b is semantically correct. All the other finite state models are semantically inconsistent as a model element can go through its possible states in exactly one possible sequence—$G_{(N)} \rightarrow G_{(C)} \rightarrow G_{(F)}$. We call this sequence the *default sequence*, and must be satisfied by all model elements. Now, let us increase the complexity by incorporating one more model element in the SR-diagram, i.e., let there exist two model elements in the SR-diagram. These two model elements can belong to the same actor or to two different actors. In either case, the complexity analysis remains the same.

Let A_1 and A_2 be two different actors, each with a single goal node G_1 and G_2, respectively. Since each goal can be in either of three states, the total number of possible combined states is 3^2 (=9). However, since both G_1 and G_2 must

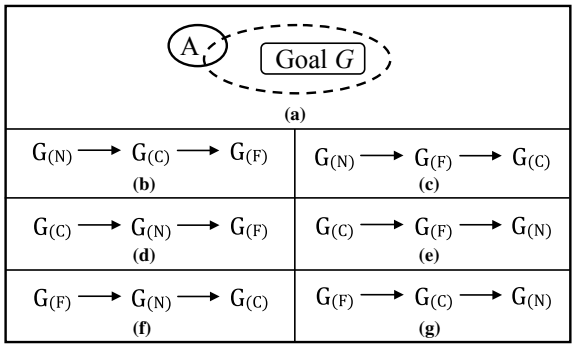

Fig. 4.3 **a** Actor A with a single goal G; **b** The only semantically correct finite state sequence; **c**–**g** Other possible finite state sequences that can be derived by permuting the state space but which are semantically incorrect

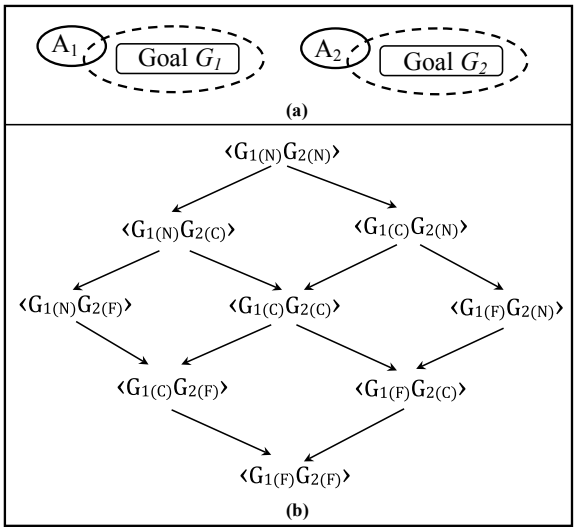

Fig. 4.4 **a** Actors A_1 and A_2 with goals G_1 and G_2, respectively; **b** The State Sequence Graph over the set of $3^2 = 9$ possible states

individually satisfy the *default sequence*, it becomes interesting to enumerate the valid state transition sequences that do not violate the *default sequence* of individual model elements. We draw a *State Sequence Graph* that captures all possible valid state transition sequences from the source node—denoted by $(G_{1(N)}\ G_{2(N)})$—to the destination node—denoted by $(G_{1(F)}G_{2(F)})$. Figure 4.4b illustrates the *State Sequence Graph* for these two goals.

The *State Sequence Graph* has all the nine possible combined state representations as vertices. These vertices are connected in the form of a lattice as all state transitions do not satisfy the *default sequence*. Each path in the *State Sequence Graph*, from the source node $(G_{1(N)}G_{2(N)})$ to the destination node $(G_{1(F)}\ G_{2(F)})$, defines a semantically valid sequence of state transitions. In other words, each such path represents a finite state sequence corresponding to the given i* model. Thus, with two model elements, we obtain six different finite state sequences that satisfy the *default sequences* of the individual model elements.

4.1.1.1 State Sequence Graph

A *State Sequence Graph*, G_{SS}, can be defined as a 2-tuple $\langle V, E \rangle$ where V represents the set of vertices and E represents a set of directed edges such that

1. Each vertex $v_i \in$ V is an n-tuple $(\bar{G}_1, \bar{G}_2, \ldots, \bar{G}_n)$ that represents the state of each of the n model elements that appear in the SR model.
2. Each directed edge $e_{ij} \in$ E is directed from vertex v_i to vertex v_j such that $v_i \rightarrow v_j$ satisfies the *default sequence* for any one of the n model elements represented in every vertex. This implies that $v_i \rightarrow v_j$ represents either of the following:
   (a) Some goal G_i goes from the *NC* state to the *CNF* state, denoted by $(\bar{G}_1 \ldots G_{i(N)} \ldots \bar{G}_n) \rightarrow (\bar{G}_1 \ldots G_{i(C)} \ldots \bar{G}_n)$, or
   (b) Some goal G_i goes from the *CNF* state to the *F* state, denoted by $(\bar{G}_1 \ldots G_{i(C)} \ldots \bar{G}_n) \rightarrow (\bar{G}_1 \ldots G_{i(F)} \ldots \bar{G}_n)$.
3. The number of vertices in the vertex set V is 3^n, i.e., $|V| = 3^n$.
4. Each path from the source vertex $(G_{1(N)} \ldots G_{n(N)})$ to the sink vertex $(G_{1(F)} \ldots G_{n(F)})$ represents a valid ordering of state transitions that satisfies the *default sequence* of the individual model elements, i.e., every unique path $(G_{1(N)} \ldots G_{n(N)}) \rightarrow .... \rightarrow (G_{1(F)} ... G_{n(F)})$ represents a finite state model.

The next level of complexity involves three different model elements. The analysis remains the same irrespective of how these three model elements are distributed between actors. Let G_1, G_2 and G_3 be the three goals modelled in the SR-diagram. As mentioned above, since each goal can be in either of three states, this particular situation will result in a state space with $3^3 (=27)$ combined states. The *State Sequence Graph* can be obtained as shown before. A detailed reachability analysis yields 90 different paths that exist between the source vertex $(G_{1(N)} G_{2(N)} G_{3(N)})$ and the sink vertex $(G_{1(F)} G_{2(F)} G_{3(F)})$. Each of these paths represents a sequence of valid state transitions such that none of the three goals G_1, G_2, and G_3 violate the *default sequence*. Thus, with three model elements in the SR-diagram we get 90 possible finite state sequences that correspond to the given i* model.

4.1.1.2 Counting Multidimensional Lattice Paths

In general, it is interesting to observe how the number of paths within a *State Sequence Graph* increases in accordance to the number of model elements (k) within an i* model. It is intuitive from the above case studies that the growth of thestate space size can be represented as an exponential function $f(k) = 3^k$. This is because each model element can exist in either of three states. On the other hand, the function representing the growth of the finite state model space is far more complex. Before going into the details of evaluating an upper bound for the finite state model space, we need to keep in mind that every model element is initially in the *Not Created* state and it needs two transitions to reach the *Fulfilled* state. Thus, the distance covered by each model element is always 2.

Consider the case where $k = 2$. Since each model element needs to cover a distance of 2, we can consider $P_{1(N)}P_{2(N)}$ and $P_{1(F)}P_{2(F)}$ as the *Least Upper Bound* and the *Greatest Lower Bound* of a 2×2 lattice. In general, the number of paths within a $n_1 \times n_2$ lattice is given by

$$L_P = \binom{n_1 + n_2}{n_1} = \frac{(n_1 + n_2)!}{n_1! n_2!} \tag{4.1}$$

So for a 2×2 lattice structure, we have

$$L_P = \binom{2+2}{2} = \frac{(2+2)!}{2!\,2!} = \frac{4!}{2!\,2!} = \frac{24}{4} = 6.$$

This is exactly what we obtain from our example of Fig. 4.4.

When $k = 3$, we can represent the set of all possible state sequences from $P_{1(N)}P_{2(N)}P_{3(N)}$ to $P_{1(F)}P_{2(F)}P_{3(F)}$ as a 3-dimensional cubic lattice with each dimension having distance 2. In general, the number of paths in a 3-dimensional cubic lattice with dimensions (n_1, n_2, n_3) is given by

$$L_P = \binom{n_1 + n_2 + n_3}{n_1, n_2, n_3} = \frac{(n_1 + n_2 + n_3)!}{n_1! n_2! n_3!} \tag{4.2}$$

So for a 3-dimensional cubic lattice with dimensions (2, 2, 2), we have-

$$L_P = \binom{2+2+2}{2, 2, 2} = \frac{(2+2+2)!}{2!\,2!\,2!} = \frac{6!}{2!\,2!\,2!} = \frac{720}{8} = 90.$$

Again, this is exactly what we obtain from our previous case study.

To generalize an upper bound on the growth function of the finite state model space, we need to realize that for k different model elements in the i* model we need a k-dimensional hypercube lattice. The number of paths in such a k-dimensional hypercube lattice with dimensions $(n_1, n_2, \ldots, n_k)$ is given by

$$L_P = \binom{n_1 + n_2 + \cdots + n_k}{n_1, n_2, \ldots, n_k} = \frac{(n_1 + n_2 + \cdots + n_k)!}{n_1! n_2! \ldots n_k!} = \frac{(\sum_{i=1}^{k} n_i)!}{\prod_{i=1}^{k} (n_i!)} \tag{4.3}$$

Irrespective of the number of model elements in the i* model, since each model element travels a distance of 2 to become fulfilled, we have the condition $\forall_{i=1}^{k}$, $n_i = 2$. The total number of paths is, thus, given by

$$L_P = \frac{(\sum_{i=1}^{k} 2)!}{\prod_{i=1}^{k} (2!)} = \frac{(2 * \sum_{i=1}^{k} 1)!}{\prod_{i=1}^{k} (2)} = \frac{(2k)!}{2^k}. \tag{4.4}$$

4.1.1.3 The Naïve Algorithm

Input: SR-diagram of the i* model of an enterprise
Output: The set of plausible finite state sequences that can be derived from the given i* model
Data Structure: A *List* that stores all the model elements appearing in the SR model

Step-1: Select actor A_j and populate *List* with all the model elements that appear within the actor boundary of A_j.
Step-2: Repeat Step-1 for all actors and proceed to create the *State Sequence Graph*.
Step-3: Initialize the vertex set V with the vertex $(P_{1(N)} \ldots P_{n(N)})$ representing all model elements in the `Not Created` state.
Step-4: Select a vertex v_i from the vertex set V.
Step-5: Create a new vertex v_i' at a distance of 1 from v_i such that

(a) Some model element P_k makes a transition from `Not Created` to `Created Not Fulfilled` state, i.e., $(\bar{P}_1 \ldots P_{k(N)} \ldots \bar{P}_n) \rightarrow (\bar{P}_1 \ldots P_{k(C)} \ldots \bar{P}_n)$, OR
(b) Some model element P_k makes a transition from `Created Not Fulfilled` to `Fulfilled` state, i.e., $(\bar{P}_1 \ldots P_{k(C)} \ldots \bar{P}_n) \rightarrow (\bar{P}_1 \ldots P_{k(F)} \ldots \bar{P}_n)$.

Step-6: If $v_i' \notin V$, then $V = V \cup v_i'$.
Step-7: Repeat Steps 4–6 till the vertex set V is not filled, i.e., while $|\mathrm{V}| < 3^n$.
Step-8: Select any two vertices v_i,v_j from the vertex set V that are separated by a distance of 1.
Step-9: Set up a directed edge from v_i to v_j if and only if $v_i \rightarrow v_j$ satisfies the *default sequence* for any one of the n model elements.
Step-10: Repeat Steps 8–9 till we obtain the n-dimensional hypercube lattice structure (*State Sequence Graph*) for the SR model.
Step-11: Each path from the vertex $(P_{1(N)} \ldots P_{n(N)})$ to the vertex $(P_{1(F)} \ldots P_{n(F)})$ represents a finite state sequence that corresponds to the given SR model.
Step-12: Stop.

4.1.1.4 Simulation Results: The Hyperexponential Explosion

Equation 4.4 of Sect. 4.1.1.2 can be used to generate a data set and observe how the state space and the finite state model space grows with increasing number of model elements in the i* model. Table 4.1 represents such a data set with the number of model elements increasing from 5 to 85 in steps of 5. Data thus obtained have been plotted on a graph and the trends are observed. Figure 4.5 depicts the rate of growth for both the state space and the finite state model space with respect to the number of model elements appearing in the given i* model.

Interpretation of the graph is quite interesting. Both the growth curves plotted in Fig. 4.5 appear to be somewhat linear in nature, although they are not straight lines.

Table 4.1 Rate of growth of space w.r.t. the number of model elements

No. of process elements	State space	Finite state model space
5	243	113400
10	59049	2.37588E+15
15	14348907	8.09487E+27
20	3486784401	7.78117E+41
25	8.47289E+11	9.06411E+56
30	2.05891E+14	7.74952E+72
35	5.00315E+16	3.48622E+89
40	1.21577E+19	6.5092E+106
45	2.95431E+21	4.2227E+124
50	7.17898E+23	8.289E+142
55	1.74449E+26	4.4083E+161
60	4.23912E+28	5.8022E+180
65	1.03011E+31	1.7528E+200
70	2.50316E+33	1.1403E+220
75	6.08267E+35	1.5123E+240
80	1.47809E+38	3.8999E+260
85	3.59175E+40	1.876E+281

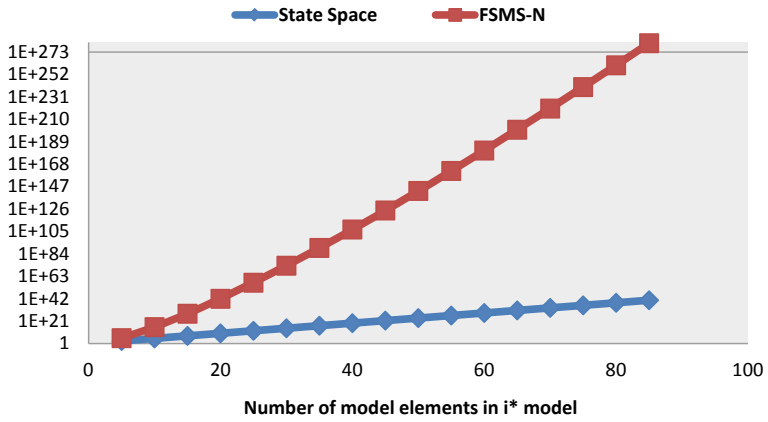

Fig. 4.5 Graph depicting the rate of growth of the state space and finite state model space with respect to the number of model elements in the i* model for the Naïve Algorithm

A careful analysis of the graph reveals that the vertical axis is a *logarithmic scale* where the values represent exponentially increasing integers. These values range from 1 to $1.876E + 281$. Thus, although the curves appear to be somewhat linear, they represent exponential growth functions on the logarithmic scale. In fact, the state space growth function, as represented by the blue curve, actually represents the growth function $f(k) = 3^k$. The growth function of the finite state model space, as represented by Eq. 4.4, is shown by the red curve.

The most significant inference that can be drawn from the graph is that the gradient of the blue curve is much less compared to that of the red curve. The gradient of a linear curve on a logarithmic scale signifies the rate of growth of the exponential function. This implies that although both the state space and the finite state model space grow exponentially, the rate of growth of the finite state model space is significantly large compared to that of the state space. In fact, the values in Table 4.1 reveal that, in every step, the state space grows by an approximate factor in the range $(10^2, 10^3)$, whereas the finite state model space grows by an approximate factor in the range $(10^{19}, 10^{20})$. This is really huge in terms of the rate of growth.

This extremely rapid growth in size of the finite state model space, caused by the *Naïve Algorithm*, results in a *hyperexponential explosion*. The growth curve of the finite state model space is so steep that it reaches infinitely large values for quite small number of model elements in the i* model. This implies that the finite state model space becomes quite unmanageable in real time when we are looking at the i* model of an entire enterprise. Thus, it becomes necessary to tackle this explosion in the finite state model space. One of the means to control this undesirable explosion is to extract partial sequence information that remains embedded within an i* model and perform some pruning activities while the finite state models are being generated. The *Semantic Implosion Algorithm* is proposed in the next section with this same intent. The proposed solution provides a significant improvement in terms of the rate of growth of the finite state model space.

4.1.2 The Semantic Implosion Algorithm (SIA)

The motive here is to prevent the *hyperexponential explosion* of the finite state model space that is caused by the *Naïve Algorithm*. Although the *Naïve Algorithm* generates all possible finite execution traces that can be derived from an i* model, some sort of pruning can be done on this model space. The simplest means of doing this is to feed the derived FSM into some standard model checker like *NuSMV* and check the model against user-defined temporal compliance rules, specified using CTL or LTL. However, since this needs to be done on the entire finite state model space, the time complexity of the entire process becomes unmanageable even when machine-automated.

It is desirable to prevent the *hyperexponential explosion* from occurring in the first place. We propose the *Semantic Implosion Algorithm*, or *SIA*, that tries to achieve this and proves to be successful to a good extent. *SIA* is based on the underlying hypothesis

that although an i* model is sequence agnostic, there exists some embedded temporal information that can be extracted and exploited to reduce the plausible space of finite state models. Temporal compliance rules may be further defined to reduce the size of the finite state model space.

Every model element P_i residing within the SR-diagram of an actor is uniquely identified using a system variable V_i. Every system variable V_i can have either of three values—0, 1, or 2—representing the `Not Created` ($P_{i(N)}$), `Created Not Fulfilled` ($P_{i(C)}$) and `Fulfilled` ($P_{i(F)}$) states, respectively. Every time a new model element P_j is encountered, a corresponding system variable V_j is created and initialized to 0 representing the Not Created state. This is reflected in the finite state model of the enterprise with a transition from the current state to a new state where the corresponding system variable V_j becomes a member of the state variable list.

The algorithm proceeds to explore the children of a chosen model element P_i. Before doing so, the corresponding system variable V_i is changed from 0 to 1 and pushed onto a *stack*. This is reflected in the finite state model with a state transition from the current state to a new state that reflects the fact that P_i has been created but not fulfilled. A model element is said to be `Fulfilled` when either it has no children (we have reached the actor boundary) or all its child model elements have been individually fulfilled. When this happens, the system variable V_i, corresponding to the model element P_i, is popped from the *stack* and updated with the value 2. A corresponding state transition is incorporated in the finite state model that reflects the fact that model element P_i has been fulfilled. Figure 4.6 illustrates the finite state model corresponding to a single model element and how the corresponding system variable is incorporated and updated along each transition.

However, it is interesting to note how the child model elements of a particular parent are processed. The processing differs for *task decompositions* and *means-end decompositions*. A *task decomposition* is an AND-decomposition and demands that all the child model elements be fulfilled in order to declare that the parent has also been fulfilled. A *means-end decomposition*, on the other hand, is an OR-decomposition and provides alternate strategies to fulfil the parent model element. Let us elaborate on the consequences of these two decompositions.

A *task decomposition* requires that all the child model elements be fulfilled before changing the state of the parent model element to the fulfilled state. However, since an i* model is sequence agnostic, the child model elements may be fulfilled in any random order. System variables associated with the child model elements should not defy the *default sequence* defined in Sect. 4.1.1. Let a model element P_j be decom-

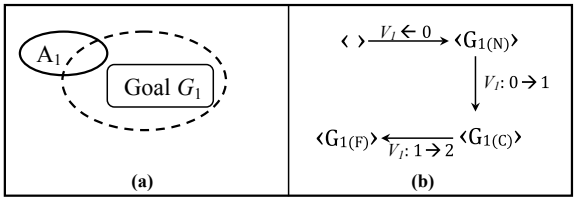

Fig. 4.6 **a** Actor A_1 with goal G_1; **b** The corresponding finite state model

posed by a *task decomposition* to a set of model elements $\langle P_1, P_2, \ldots, P_m \rangle$. The system variables associated with these model elements are V_1, V_2, …, V_m, respectively. We define a state transition from the current state with $V_j = 1$ to a new state with the state variables $V_j = 1, \forall_{r=1}^{m}$, $V_r = 0$. There exists several execution sequences of the decomposed model elements that finally results in a state with the state variables, $V_j = 1, \forall_{r=1}^{m}$, $V_r = 2$. The set of all possible execution sequences can be defined using a lattice structure, similar to the one shown in Fig. 4.4. Since all child model elements are fulfilled, we define another state transition in the finite state model that reflects the fact that the parent model element is also fulfilled, i.e., the new state has state variables V_j=2. The finite state model corresponding to such a *task decomposition* is shown in Fig. 4.7.

The interpretation of the figure is quite interesting. The lattice structure represents the set of all possible execution sequences that result in the successful fulfilment of the task decomposition. As seen in Sect. 4.1.1, the number of paths in a lattice structure for two model elements is 6. All of these 6 paths represent valid execution sequences or state transitions. Each path gives rise to a different finite execution sequence. This

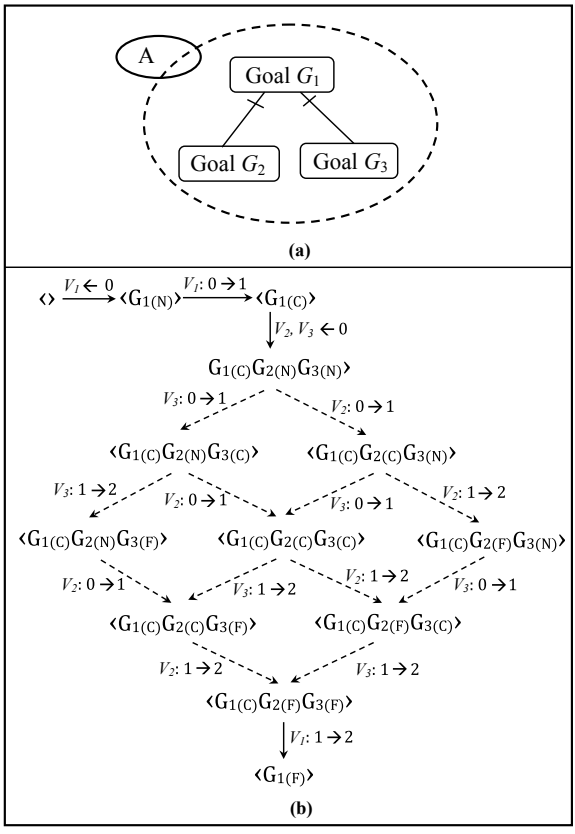

Fig. 4.7 **a** Actor A_1 with goals G_1, G_2 and G_3 connected through a task decomposition; **b** The corresponding set of all possible finite state models captured in a state sequence graph

implies that the task decomposition shown in Fig. 4.7 gives rise to 6 possible finite state sequences. The *Naïve Algorithm*, on the other hand, would generate a lattice structure with three model elements and the number of possible finite state sequences would become 90. This is a significant reduction in the finite state model space. In fact, the significant observation here is that a lattice structure will be generated only where task decompositions take place. In other words, only task decompositions will increase the size of the finite state model space.

A *means-end* decomposition is easier to handle. OR-decompositions, in general, do not increase the size of the finite state model space. Rather, if a particular model element P_j decomposes via a *means-end decomposition* into k model elements $\langle P_1, P_2, \ldots, P_k \rangle$, then we introduce k different transitions from the current state ($V_j = 1$) to k unique new states, each representing one of the k alternate means ($V_j = 1, V_p = 0, \forall_{p=1}^{k}$). An OR-decomposition is characterized by the fact that fulfilling any one of the alternate means implies fulfilling the parent model element. Thus, each of these k new states will make two transitions (labelled by V_p:0→1 and V_p:1→2) to reach their respective fulfilment states. Each alternate means will have a separate fulfilment state labelled by $V_j = 1, V_p = 2, \forall_{p=1}^{k}$. All the k fulfilment states will converge to a final state that represents the fulfilment of the parent model element P_j and is labelled by $V_j = 2$. The structure obtained is similar to the longitudinal lines on the globe of the earth. Figure 4.8 illustrates this further.

4.1.2.1 Some Interesting Features

1. Decompositions can be *nested*. This implies that decompositions can occur within other decompositions. One particular decomposition link may be further blown up with a second decomposition. For instance, *means-end decompositions* may be followed by a *task decomposition* along one means-end link and a *means-end decomposition* along some other means-end link. Figure 4.9 illustrates this scenario. This nesting of decompositions does not require any modifications on the algorithm. The corresponding finite state model is built accordingly where the state subsequences of the lower level decomposition is mereologically connected to the finite state model of the higher level decomposition.
2. It is interesting to note what happens if we reach a model element G_3, located at the actor boundary of actor A_1, that is dependent on some model element G_4 that is located at the actor boundary of actor A_2 (refer Fig. 4.10a).
   In this situation, we first proceed to complete the finite state models of the individual actors. We assume that the dependency between model elements G_3 and G_4 will be satisfied and pop out the system variable V_3 from the *stack* to set its value to 2. At the same time, we introduce a *temporary transition* in the corresponding finite state model that changes the state of G_3 from Created Not Fulfilled (*CNF*) to Fulfilled (F). This is shown in Fig. 4.10b. Such an assumption is necessary to proceed with the construction of the finite state model of individual actors.
   We need to maintain a list of all such dependencies. A *Global List* is maintained that stores 2-tuples of the form $\langle dependervariable, dependeevariable \rangle$.

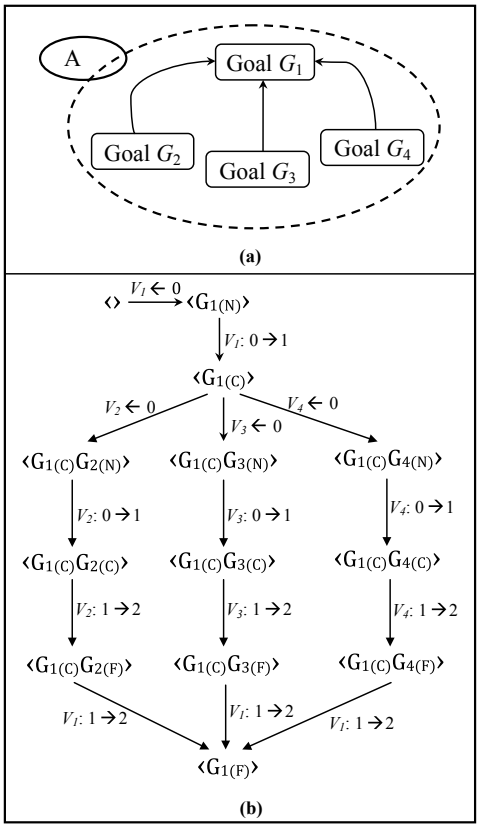

Fig. 4.8 **a** Actor A_1 with goals G_1, G_2, G_3 and G_4 connected through a means-end decomposition; **b** The corresponding finite state model

Once the finite state models of the individual actors have been built, the elements of the *Global List* are accessed. The above dependency has an entry of the form $\langle V_{13}, V_{24}\rangle$ and is interpreted as model element G_3 within actor A_1 depending on actor A_2 for model element G_4. The *temporary transition* in the finite state model of actor A_1 representing the change V_3: 1 $\rightarrow$ 2 is replaced by two new transitions that connect the finite state models of actors A_1 (FSM$_1$) and A_2 (FSM$_2$). The first transition is established from the state in FSM_1 having label $V_3 = 1$ to the state in FSM_2 having label $V_4 = 2$. The second transition is placed from the state in FSM_2 having label $V_4 = 2$ to the state in FSM_1 having label $V_3 = 2$. $\langle V_{13}, V_{24}\rangle$ is removed from the *Global List*. Figure 4.10c illustrates this process.

3. Dependency resolution causes state transitions to be set up between states belonging to the finite state models of the *depender* and the *dependee*. If the *depender* and *dependee* have M and N possible finite state sequences in their models, respectively, then we get $M \times N$ combination of sequences for interlinking the finite state models of the *depender* and *dependee*. The dependency resolution is reflected in all the $M \times N$ combinations.

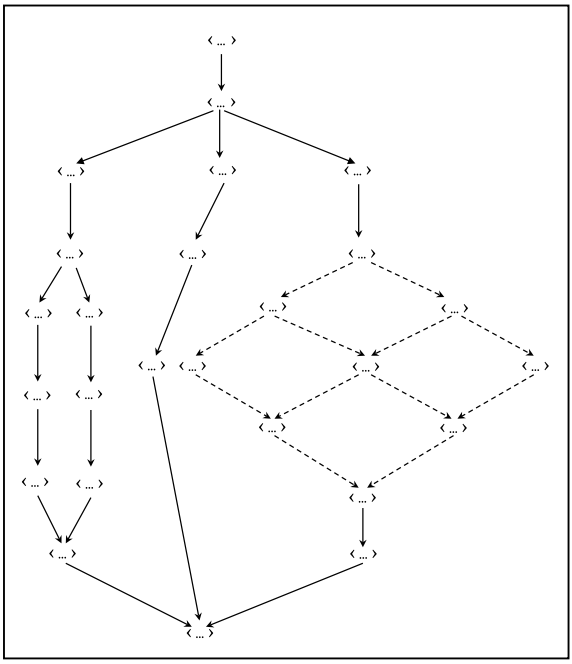

Fig. 4.9 The state sequence graph corresponding to a nested decomposition. A higher level means-end decomposition contains another means-end decomposition along the leftmost link and a task decomposition along the rightmost link

Let n be the total number of model elements occurring in the SR-diagram of the enterprise. The terminating condition of the *Semantic Implosion Algorithm* is given by the constraint, $\forall_{j=1}^{n}$, $V_j = 2$, the *stack* is empty and the *Global Dependency List* is empty. The algorithm initiates with the root model elements at the actor boundaries. State transitions are defined in the corresponding finite state model as and when model elements are discovered, explored and fulfilled. Let us look into the *Semantic Implosion Algorithm* now.

4.1.2.2 The Semantic Implosion Algorithm

Input: SR-diagram of the i* model of an enterprise.
Output: The finite state model that can be derived from the given i* model containing the set of plausible finite state sequences.
Data Structure: A *Local Stack* for each actor that stores model elements of the actor and a *Global List* to keep track of dependencies between actors.

Step-1: For every model element P_i that is not at the end of a *task decomposition* or *means-end* link, assign a system variable $V_i = 0$. Perform a *Depth-First Scan* of the SR-diagram of each actor starting at these boundary model elements.

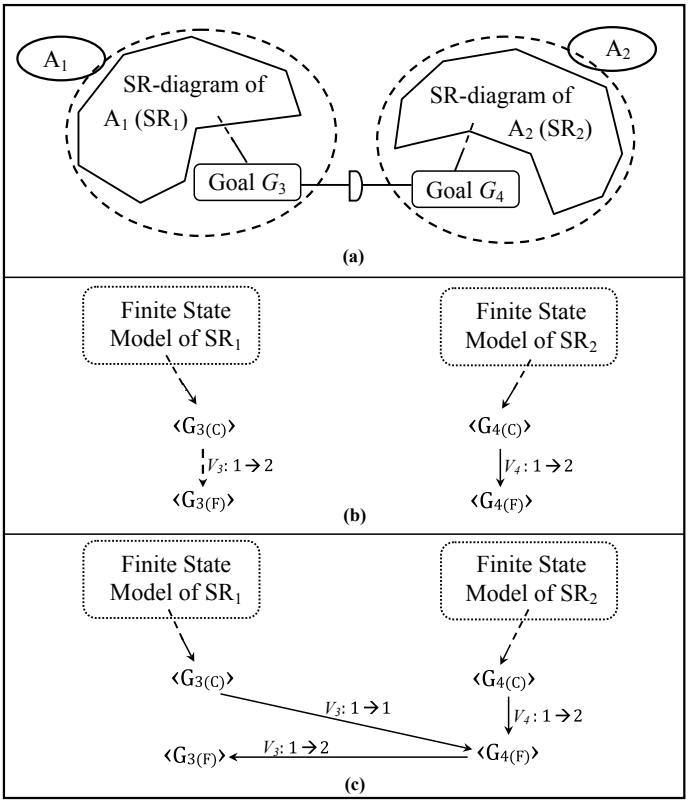

Fig. 4.10 **a** Goal G_3 of actor A_1 dependant on Goal G_4 of actor A_2; **b** Temporary transition from $G_{3(C)}$ to $G_{3(F)}$ introduced; **c** Resolution of the dependency by replacing the temporary transition with two permanent transitions

Step-2: For any model element P_j with $V_j = 0$, set $V_j = 1$ and push it onto the *Local Stack*. Reflect this transition in the finite state model by plotting a transition from the `Not Created` state to the `Created Not Fulfilled` state. Label this transition V_j:0→1.

Step-3: Discover all model elements $\langle P_1, P_2, \ldots, P_q \rangle$ that stem from the element P_j and are connected to P_j with *task decomposition* or *means-end* links. For each such element P_k, initialize a system variable V_k such that $\forall_{k=1}^{q} V_k = 0$.

(a) If P_j is at an actor boundary with no elements stemming from it and with no dependencies to other actors, pop V_j from the *Stack* and set $V_j = 2$. Set up a corresponding transition in the finite state model from the `Created Not Fulfilled` state to the `Fulfilled` state. Label this transition V_j:1→2.

(b) If P_j is dependent on some other actor for fulfilment, then pop V_j and insert it into the *Global List* with value $V_j = 2$. Insert a *tem-*

porary transition between states `Created Not Fulfilled` and `Fulfilled` for element P_j. No need to label this transition as it is a *temporary transition*.

(c) If P_j undergoes a *task decomposition* then we obtain several different finite state sequences for the *task decomposition* by permuting the order of execution of the child model elements. Each such permutation can be considered to be a valid execution trace and can be attached to the overall finite state model to obtain the unique finite state model for that actor.

(d) If P_j undergoes a *means-end decomposition* then we obtain multiple transitions from the current node in the same finite state model. Each transition represents an alternate strategy and is triggered by the corresponding guard condition. All the alternate state transitions emanating from the parent model element must converge at a state that represents that the parent model element has been fulfilled.

Step-4: Repeat *Steps 2–3* for all siblings of P_j in all the finite state models generated for actor A_i.

Step-5: Repeat *Steps 1–4* until the *Local Stack* is empty. This leaves us with the set of plausible finite state models of an actor A_i.

Step-6: Repeat *Steps 1–5* to extract all the possible finite state models of all the actors in the i* model.

Step-7: Remove elements of the form $\langle V_{ik}, V_{jl} \rangle$ from the *Global List*.

Step-8: Remove the *temporary transitions* corresponding to V_{ik} from the finite state model of actor A_i.

Step-9: Insert transitions from the P_k-`Created Not Fulfilled` state in the finite state model of actor A_i to the P_l-`Fulfilled` state in the finite state model of actor A_j. Label these transitions $V_k{:}1 \rightarrow 1$.

Step-10: Insert another set of transitions from the P_l-`Fulfilled` state to the P_k-`Fulfil- ed` state between the finite state models of actors A_i and A_j. Label these transitions $V_k{:}1 \rightarrow 2$.

Step-11: Repeat *Steps 7–10* until the *Global List* is empty and all the dependencies have been resolved.

Step-12: Stop.

4.1.3 Soundness and Completeness

Both the *Naïve Algorithm* and the *Semantic Implosion Algorithm* are *complete* because given a goal model both the algorithms are capable of generating a finite state model which include all the possible state transitions for valid execution traces. However, it is not wise to say that the *Naïve Algorithm* is *sound* because the generated finite state model also contains invalid state transitions. The *Semantic Implosion Algorithm*, on the hand, is *sound* because it contains all the valid transitions within the finite state model.

4.2 Complexity Analysis

Let us perform some analytics on comparing and contrasting the heuristics of the *Naïve Algorithm* and the *Semantic Implosion Algorithm*. The two metrics that are used for this analysis are the *State Space* (SS) and the *Finite State Model Space* (FSMS). However, since both algorithms share the concept of every model element going through 3 states, the SS metric will be the same for both algorithms and is defined by the function $f(k) = 3^k$, where k represents the number of model elements in the given SR-model. The FSMS metric is far more crucial in contrasting the heuristics that underline the two algorithms.

Figure 4.5 of Sect. 4.1.1.4 clearly illustrates the *hyperexponential explosion* caused by the *Naïve Algorithm* in the finite state model space. This is mainly due to the fact that the *Naïve Algorithm* considers all possible orderings of the model elements while ensuring the *default sequence* of each individual model element. A careful understanding of the *Semantic Implosion Algorithm* reveals that, while the finite state models of individual actors are being built, the finite state model space increases only when the following conditions hold:

1. Whenever a *nested Task Decomposition* is encountered. Suppose a goal/task is decomposed to k different model elements. Since an i* model is sequence agnostic, these k model elements can be executed in any order. The set of all possible execution traces is given by a k-dimensional hypercube lattice with each dimension having distance 2. As discussed in Sect. 4.1.1.2, the finite state model space increases by a factor of $\frac{(2k)!}{2^k}$ as given by Eq. 4.4. This implies that if the finite state model space already has p execution traces, a Task Decomposition into q model elements causes the size of the finite state model space to become p. $\frac{(2q)!}{2^q}$. In general, if the SR-diagram of an actor within the i* model has D_T task decompositions, and the number of possible alternate execution sequences generated by each of these task decompositions be given by $\#Seq_1, \#Seq_2, \ldots, \#Seq_{D_T}$, then the finite state model space size is given by the following relation:

$$S = \prod_{i=1}^{D_T} \#Seq_i \tag{4.5}$$

2. Whenever a dependency is being resolved. *Dependency resolution* results in merging the finite state model space of the *depender* and the *dependee*. If the finite state model spaces of actors A_i and A_j contain M and N finite state sequences, respectively, and there exists at least one dependency between these actors, then irrespective of the number of dependencies between A_i and A_j, the size of the finite state model space changes from $M + N$ to $M \times N$. Again, if actor A_j requires dependency resolution with actor A_k, and actor A_k has L finite state models, then the combined finite state model space has size $L \times M \times N$.
3. Let there be n actors participating in an i* model. Let the size of the finite state model spaces of the individual actors be given by $S_1, S_2, \ldots, S_n$, respectively.

Assuming that all the actors are interconnected with dependencies, the finite state model space (*FSMS*) for the entire enterprise is given by the following equation:

$$FSMS = \prod_{i=1}^{n} S_i \tag{4.6}$$

Both *Dependency Resolution* and *nested Task Decomposition* conditions are represented using the cartesian product relation. So, performance analysis of the two heuristics boils down to two basic steps. The first step involves observing the growth of the finite state model space for each individual actor. The second step is to observe the growth of the finite state model space for the entire enterprise.

4.2.1 Actor Internal Analytics

It is very difficult to predict the distribution of model elements within the SR-diagrams of individual actors. Since this is the first step of behaviour analysis, we are concerned with the growth of the finite state model space for individual actors within an i* model. In order to generate a consistent data set, we assume a uniform distribution of model elements. We increase the number of model elements occurring within the SR-diagram of an actor in the i* model, in steps of 5. Without loss of *uniformity*, we assume that for every 5 model element within an actor, there exists a task decomposition of 4 elements. This assumption is necessary as we want to estimate an upper bound on the growth function and the number of finite state sequences grows significantly with Task Decompositions as opposed to Means-End Decompositions.

We know that the *Naïve Algorithm* causes the finite state model space to grow according to Eq. 4.4, i.e., FSMS-N = $\frac{(2k_1)!}{2^{k_1}}$, where k_1 is the number of model elements in the i* model. The *Semantic Implosion Algorithm* grows only on the basis of task decompositions. The number of possible execution sequences generated by a 4-element task decomposition is obtained by substituting $k = 4$ in Eq. 4.4, i.e., $\frac{(2\times4)!}{2^4} = \frac{8!}{16} = 2520$. Since every 4-element task decomposition increases the finite state model space size by a factor of 2520, applying the cartesian product relation, we obtain the growth function of the *Semantic Implosion Algorithm* to be given by FSMS-S = 2520^{k_2}, where k_2 is the number of 4-element task decompositions occurring within the SR-diagram of an actor. Table 4.2 reflects such a data set.

The performance ratio parameter in Table 4.2 represents the reduction in finite state model space, obtained by the *Semantic Implosion Algorithm*, with respect to the *Naïve Algorithm*. The smaller the ratio the greater is the reduction in finite state model space achieved by the *Semantic Implosion Algorithm*. As the values in this column reflect, the reduction rate is not constant and increases from $\Theta(10^{-9})$ to $\Theta(10^{-17})$. This is also evident from the graph plotted for this data.

The graph plotted on the basis of this data is shown in Fig. 4.11. It is interesting to analyse the graph. The vertical axis is again a logarithmic scale of integers. The almost

Table 4.2 Actor internal analytics

No. of process elements (k_1)	No. of task decompositions (k_2)	Naïve algorithm	SI algorithm	Performance ratio
		FSMS-N = $\frac{(2k_1)!}{2^{k_1}}$	FSMS-S = 2520^{k_2}	($\frac{FSMS\text{-}S}{FSMS\text{-}N}$)
5	1	113400	2520	0.0222
10	2	2.37588E+15	6350400	2.67286E−9
15	3	8.09487E+27	1.6E+10	1.97656E−18
20	4	7.78117E+41	4.03E+13	5.17917E−29
25	5	9.06411E+56	1.02E+17	1.12532E−40
30	6	7.74952E+72	2.56E+20	3.30343E−53
35	7	3.48622E+89	6.45E+23	1.85041E−66
40	8	6.5092E+106	1.63E+27	2.50415E−80
45	9	4.2227E+124	4.1E+30	9.70943E−95
50	10	8.289E+142	1.03E+34	1.24261E−109
55	11	4.4083E+161	2.6E+37	5.89796E−125
60	12	5.8022E+180	6.56E+40	1.1306E−140
65	13	1.7528E+200	1.65E+44	9.41351E−157
70	14	1.1403E+220	4.16E+47	3.64816E−173
75	15	1.5123E+240	1.05E+51	6.94306E−190
80	16	3.8999E+260	2.64E+54	6.7694E−207
85	17	1.876E+281	6.67E+57	3.55544E−224

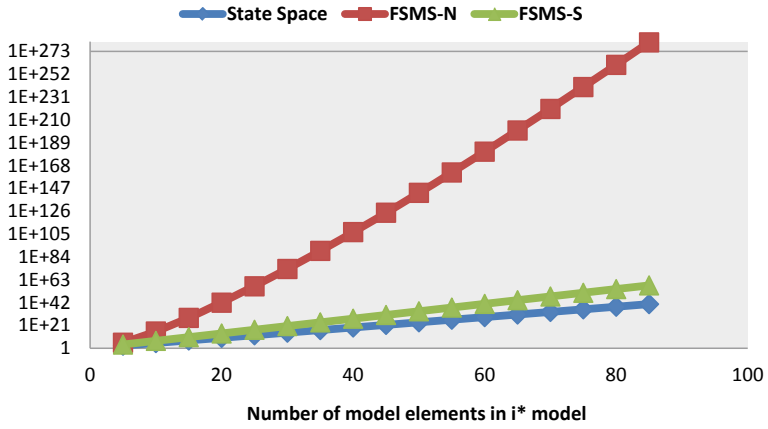

Fig. 4.11 Behaviour analysis with respect to the finite state model space of individual actors for the *Naïve Algorithm* (`FSMS-N`) and the *Semantic Implosion Algorithm* (`FSMS-S`) as the number of model elements in the i* model varies

linear curves plotted on this scale represent exponential functions. The gradient of these approximately linear curves represent the rate of growth of the corresponding exponential function. The following observations can be concluded from the graph:

1. The *blue curve* depicts the growth of the state space and is consistent for both scenarios, given by 3^k. As both algorithms have the same underlying basis that every model element goes through three states, the state space growth remains the same for both the algorithms.
2. The *green curve* represent the behaviour of the *Semantic Implosion Algorithm*. The two lines with triangle and diamond annotations are very close to each other and have almost similar gradients. This implies that the rate of growth of the finite state model space, as observed from the *Semantic Implosion Algorithm*, is almost similar to the rate of growth of the state space.
3. The *red curve* depicts the finite state model space growth of the *Naïve Algorithm*. The slope of this line is much greater than those of the green and blue lines. This represents the *hyperexponential explosion* that is a characteristic of the *Naïve Algorithm*.
4. A closer look at the FSMS values in Table 4.2 reveals the fact that the FSMS metric increases by a factor in the range of $(10^{19}, 10^{20})$, for the *Naïve Algorithm*, whereas, for the *Semantic Implosion Algorithm*, the FSMS metric increases by a factor of 10^3.

From the above data set—Table 4.2 and Fig. 4.11—it is evident that the *Semantic Implosion Algorithm* provides a huge improvement with respect to the rate of growth of the finite state model space for individual actors in comparison to the *Naïve Algorithm*. This is the significant contribution of the heuristic proposed in the *Semantic Implosion Algorithm*.

4.2.2 Inter-Actor Analytics

These analytics provide an insight into how *Actor Internal Analytics* scales up and impacts the growth rate of the finite state model space with respect to the entire i* model representing an enterprise. There are two events that impact *Inter-Actor Analytics* as follows:

1. *Density of Actors* participating in the i* model, and
2. *Distribution of Model Elements* within the SR-diagram of the actors.

Let us individually analyse how these two parameters effect the growth rate of the finite state model space.

4.2.2.1 Variation of Actor Density

In order to simulate a data set, we assume a uniform density of five model elements within individual actors and evaluate the rate of growth of the finite state model space. Similar to the data in Table 4.1, we assume that every actor has a 4-element task decomposition. The *Naïve Algorithm* does not take the semantics of the model elements into consideration and, thus, the finite state model space size can be evaluated by replacing $k = 5$ in Eq. 4.4. The finite state model space size of every actor is obtained as

$$\forall i, S_i = \frac{(2 \times 5)!}{2^5} = \frac{10!}{32} = 113400.$$

Replacing this value of S_i in Eq. 4.6, we get the finite state model space for the entire enterprise (FSMS-N) as

$$FSMS\text{-}N = (113400)^n \tag{4.7}$$

The *Semantic Implosion Algorithm*, on the other hand, causes the finite state model space of individual actors to grow only when task decompositions are encountered. Since we assume a 4-element task decomposition to exist in each actor, the finite state model space(S_i) of all the actors remains constant and is given by replacing $k = 4$ in Eq. 4.4. Thus,

$$\forall i, S_i = \frac{(2 \times 4)!}{2^4} = \frac{8!}{16} = 2520.$$

Since uniform distribution of model elements has been assumed, replacing this value of S_i in Eq. 4.6 gives the finite state model space for the entire enterprise (FSMS-S) as generated by the *Semantic Implosion Algorithm*. Thus,

$$FSMS\text{-}S = (2520)^n \tag{4.8}$$

In order to generate a simulated data set, we restrict the number of model elements in each actor to 5 and increase the density of actors (n) within the i* model of the enterprise from 5 to 55 in steps of 5. The data set is obtained by replacing these values of n in Eqs. 4.7 and 4.8. Table 4.3 represents such a data set. The performance ratio column represents the relative decrease in the finite state model space that is obtained by the *Semantic Implosion Algorithm*. Figure 4.12 shows the corresponding graph structure that is obtained by plotting this data.

Interpretation of the graph is quite intuitive. The *blue curve* represents the growth function of the *Naïve Algorithm*. In this data set, it represents the exponential function $(113400)^n$. The *red curve* plots the growth function of the *Semantic Implosion Algorithm* and represents the exponential $(2520)^n$. With the vertical axis representing a logarithmic scale of integers, the two functions are mapped as nearly linear curves with different gradients. The gradient of the blue curve is greater than the gradient

Table 4.3 Inter-actor analytics obtained by varying actor density

No. of actors (n)	Naïve algorithm	SI algorithm	Performance ratio
	FSMS-N = $(113400)^n$	FSMS-S = $(2520)^n$	$(\frac{FSMS\text{-}S}{FSMS\text{-}N})$
5	1.87528E+25	1.01626E+17	5.41924E−9
10	3.51666E+50	1.03277E+34	2.93679E−17
15	6.59471E+75	1.04956E+51	1.59152E−25
20	1.23669E+101	1.06662E+68	8.62479E−34
25	2.31914E+126	1.08396E+85	4.67397E−42
30	4.34902E+151	1.10158E+102	2.53294E−50
35	8.15562E+176	1.11949E+119	1.37266E−58
40	1.52940E+202	1.13768E+136	7.43872E−67
45	2.86805E+227	1.15618E+153	4.03124E−75
50	5.37840E+252	1.17497E+170	2.18461E−83
55	1.00860E+278	1.19407E+187	1.18388E−91

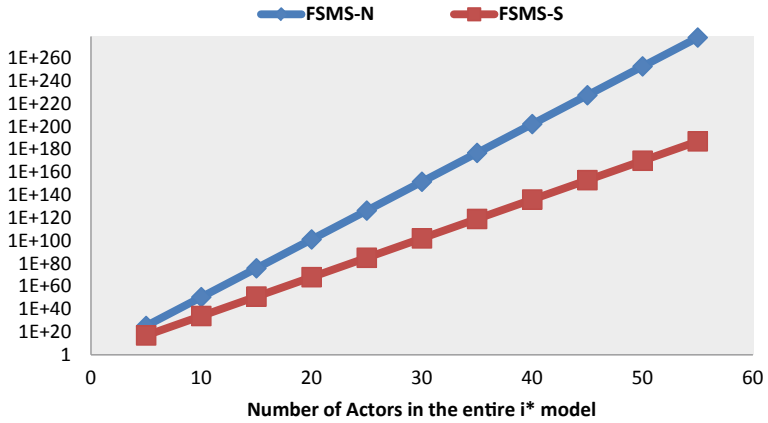

Fig. 4.12 Behaviour analysis with respect to the finite state model space of the entire enterprise for the *Naïve Algorithm* (FSMS-N) and the *Semantic Implosion Algorithm* (FSMS-S) as the density of actors in the i* model varies

of the red curve. This implies that the *Naïve Algorithm* increases the size of the finite state model space more rapidly as compared to the *Semantic Implosion Algorithm*. This is evident from the growth functions Eqs. 4.7 and 4.8 itself. However, this is an overly simplified data set with uniform distribution and semantics.

4.2.2.2 Variation of the Distribution of Model Elements

In this particular simulation, we fix the number of actors involved in the enterprise model to five. Keeping the number of actors fixed, the distribution of model elements peractor is increased from 5 to 25 in steps of 5. Assuming uniform distribution across all the actors in the i* model, every actor generates it's finite state model space with the same size. The space size changes with varying model element distribution. Let the size of the finite state model spaces of the individual actors be given by S_1, $S_2, \ldots, S_5$, respectively, for some model element distribution k.

The *Naïve Algorithm* combines Eqs. 4.4 and 4.6 to give a function representing the growth of the finite state model space as follows:

$$FSMS\text{-}N = \left(\frac{(2k_1)!}{2^{k_1}}\right)^5, \forall k_1, k_1 \in \{5, 10, 15, 20, 25\}. \tag{4.9}$$

The *Semantic Implosion Algorithm* expands the finite state model space for Task Decompositions only. Our underlying assumption that there exists a 4-element Task Decomposition for every group of 5 elements dictates the growth function of the finite state model space as follows:

$$FSMS\text{-}S = \left(\frac{(2k_2)!}{2^{k_2}}\right)^5, k_2 = k_1 \div 5, \forall k_1, k_1 \in \{5, 10, 15, 20, 25\}. \tag{4.10}$$

The data generated from Eqs. (4.9) and 4.10 is shown in Table 4.4. The number of actors has been fixed to be 5. The performance ratio values represent the improvement in finite state model space that is achieved by the *Semantic Implosion Algorithm*. The smaller the value, the greater is the gain in performance achieved by the SIA heuristics. The rapid rate of increase in performance reflects the benefits of using the improved heuristics of the *Semantic Implosion Algorithm*. Figure 4.13 represents the graph corresponding to this data.

Table 4.4 Inter-actor analytics obtained by varying the distribution of goals

No. of process elements (k_1)	Naïve algorithm	SI algorithm	Performance ratio
	FSMS-N $= (\frac{(2k_1)!}{2^{k_1}})^5$	FSMS-S $= (\frac{(2k_2)!}{2^{k_2}})^5$, $k_2 = k_1 \div 5$	$(\frac{FSMS\text{-}S}{FSMS\text{-}N})$
5	1.87528E+25	1.01626E+17	5.41924E−9
10	7.57046E+76	1.03277E+34	1.36421E−43
15	3.47576E+139	1.04956E+51	3.01966E−89
20	2.85249E+209	1.06663E+68	3.73929E−142
25	6.11823E+284	1.08399E+85	1.77174E−200

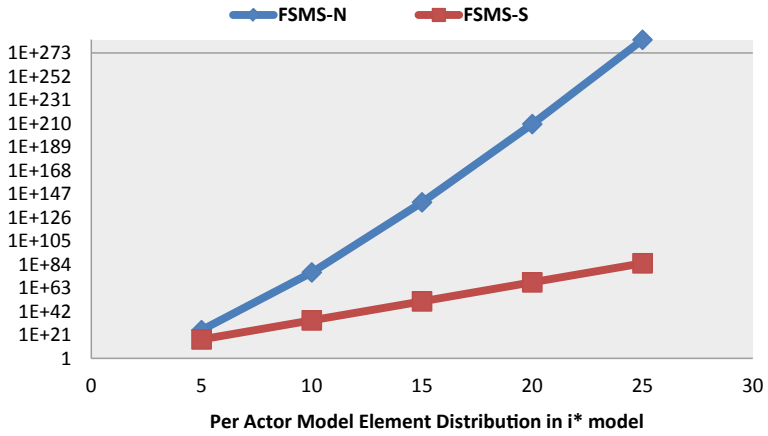

Fig. 4.13 Behaviour analysis with respect to the finite state model space of the entire enterprise for the *Naïve Algorithm* (`FSMS-N`) and the *Semantic Implosion Algorithm* (`FSMS-S`) as the distribution of model elements within actors in the i* model varies

The interpretation of the graph is quite similar to the previous graphs. The vertical axis represents a logarithmic scale of integers. Both the exponential functions, given by Eqs. 4.9 and 4.10, appear as straight lines. However, the gradients of the two lines are widely different. This implies that the rate of growth of `FSMS-N` (represented by the *blue curve*) is much greater than that of `FSMS-S` (represented by the *red curve*).

4.2.3 SIA Analytics

The analytics provided in Tables 4.2, 4.3, and 4.4, and the corresponding graphs shown in Figs. 4.11, 4.12, and 4.13, all point in the same direction. The obvious conclusion from these data sets is that the *Semantic Implosion Algorithm* provides a huge improvement over the more simple *Naïve Algorithm*. This improvement is in the context of the finite state model space and clearly establishes the superiority of the SI-heuristics in comparison to the Naïve-heuristics.

The above conclusion triggers an urge to take an insight into the behaviour of the *Semantic Implosion Algorithm* when both the parameters—*Actor Density* and *Model Element Distribution*—are varied simultaneously. Table 4.5 presents such a data set. The data is generated by varying the distribution of model elements in individual actors from 5 to 25 peractor, in steps of 5. The finite state model space size is obtained using the following equation:

$$FSMS\text{-}A = \left(\frac{(2k_2)!}{2^{k_2}}\right)^A, k_2 = k_1 \div 5 \tag{4.11}$$

Table 4.5 Inter-actor analytics obtained by varying both actor density and distribution of goals for the *Semantic Implosion Algorithm*

No. of process elements (k_1)	SI algorithm		
	FSMS-5	FSMS-10	FSMS-15
5	1.01626E+17	1.03277E+34	1.04956E+81
10	1.03277E+34	1.06662E+68	1.10158E+102
15	1.04956E+51	1.10157E+102	1.15617E+153
20	1.06663E+68	1.13769E+136	1.21349E+204
25	1.08399E+85	1.17503E+170	1.38069E+255

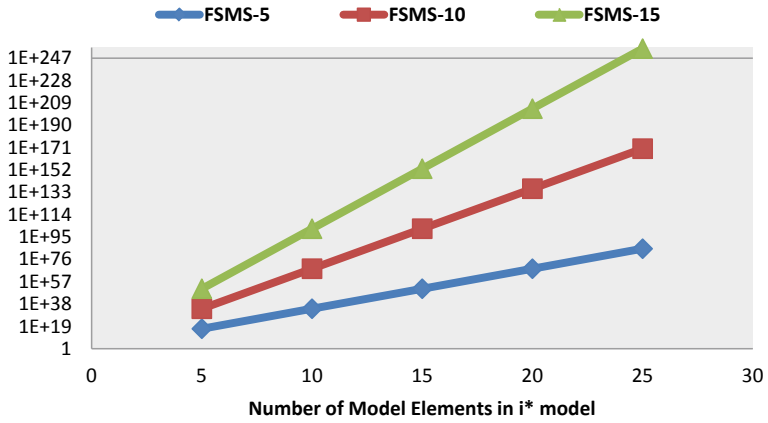

Fig. 4.14 Behaviour analysis of the *Semantic Implosion Algorithm* (w.r.t. the finite state model space) as the distribution of model elements within actors and the actor density in the i* model are both varied

Here, A represents the number of actors in the i* model of the enterprise. k_2 is obtained from k_1 based on the assumption that we have a 4-element task decomposition for every group of 5 model elements. Maintaining the uniformity of model element distribution across all the actors, we obtain the data set for 5, 10 and 15 actors, represented by FSMS-5, FSMS-10 and FSMS-15, respectively. The graph obtained from the data set in Table 4.5 is shown in Fig. 4.14.

The graph is fairly simple to analyse and interpret. The vertical axis is again a logarithmic scale. Each of the individual curves (*green, red,* and *blue*) appears to be linear but represent exponential growth functions. The fact that the finite state model space size will increase with greater number of actors has already been observed in Fig. 4.12. Hence, as the number of actors increase, the curves are positioned higher. It can also be concluded from Fig. 4.13 that for a fixed actor density, the finite state model space size increases with increasing density of model elements. Hence, the positive gradient in each of the three approximately linear curves.

The more important observation here is that the nearly linear curves are not parallel to each other. The gradient of the lines increase with increasing actor density, i.e., the green curve is steeper than the red curve which, in turn, is steeper than the blue plot. The gradient of these approximately linear curves represent the rate of growth of the exponential functions that capture the growth of the respective finite state model spaces. This means that as the actor density increases, the finite state model space increases even more rapidly.

4.3 The i*ToNuSMV Tool

Unlike dataflow and workflow models, goal models do not capture sequences of activities within the system or enterprise being designed. This makes it difficult for analysts to check the correctness of these models in the requirements phase itself. Since goal models have their own motivation, quite distinct from those of process models or workflow models, researchers have come up with completely different analysis techniques that provide new insights into the system or enterprise being developed.

Horkoff and Yu [6] have documented an exhaustive survey of the existing goal model analysis techniques and how requirement analysts can select from these alternatives based on different criteria and attributes. Applying model checking techniques to goal models (like i*) has been considered by researchers from the community. The main research problem here is that model checkers accept extended finite state models as input. Finite state models capture some sort of sequential information that represents the possible state transitions that a system can go through. Since goal models are sequence agnostic they cannot be adapted and fed into model checkers directly. We aim to provide a significant contribution in this direction by proposing the *i*ToNuSMV* tool.

The *i**ToNuSMV tool addresses this issue and performs model transformation of the given i* model. Figure 4.15 illustrates the architecture of the proposed solution. The *FSM Building* module generates a finite state model corresponding to the given i* model and the *NUSMV Mapper* module maps the generated finite state model to the NuSMV input language. The output of the *i**ToNuSMV* tool can be fed directly into the NuSMV model verifier and can be checked against temporal properties, behavioural characteristics or compliance rules written using LTL, or CTL.

*4.3.1 i*ToNuSMV Input*

The *i**ToNuSMV* prototype does not provide a graphical interface for drawing i* models. Rather, it takes a textual representation of the i*-SR-diagram as input. We use the tGRL notation as our input language. This may help in the integration of tool with the jUCMNav framework. For the i* model shown in Fig. 4.16, the corresponding tGRL representation is as follows:

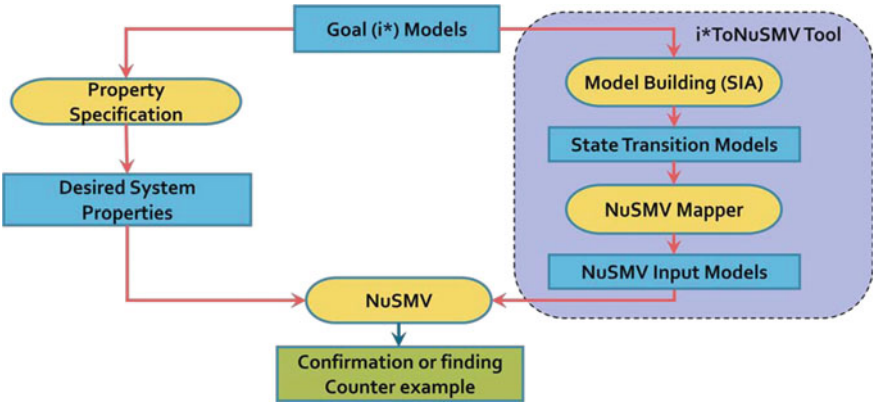

Fig. 4.15 The *i*ToNuSMV* tool

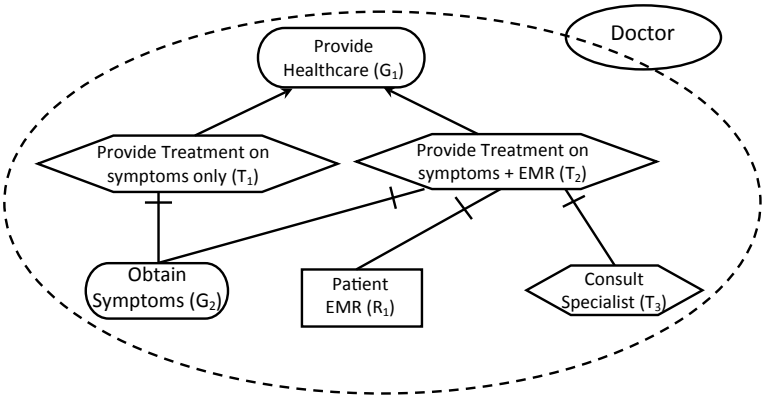

Fig. 4.16 An i* model of a single actor *Doctor*

```
grl test_model
{       actor Doctor{
            goal ProvideHealthcare{decompositionType=or;}
                task Symptoms_Treat{decompositionType=and;}
                task Symptoms_EMR_Treat{decompositionType=and;}
                    goal ObtainSymptoms{}
                    resource PatientEMR{}
                    task ConsultSpecialist{}
                Symptoms_Treat decomposedBy ObtainSymptoms;
                Symptoms_EMR_Treat decomposedBy ObtainSymptoms,PatientEMR,
                    ConsultSpecialist;
            ProvideHealthcare decomposedBy Symptoms_Treat,Symptoms_EMR_Treat;
        }
}
```

4.3.2 The Preprocessing Module

We perform a simplified lexical analysis of this textual input by tokenizing the text (using *filtokn.exe*) and then identifying keywords, operators and user-defined model artefacts (using *recognit.exe*). We proceed to identify the tree structure of the model artefacts (using *modlroot.exe*) and begin our model transformation process from the root of this tree structure.

4.3.3 The Model Transformation Module

After obtaining the desired tree structure, we proceed to generate the finite state model corresponding to the given SR-diagram. We use the *Semantic Implosion Algorithm* (SIA) [7] for converting the given i* model to a finite state model. The algorithm, as proposed by the authors, proposes a methodology for exploiting the semantics of SR-diagrams and creating a finite state model with minimum number of state transitions.

SIA uses the notion of each model artefact going through three states—*Not_Created* (NC), *Created_Not_Fulfilled* (CNF), and *Fulfilled* (F)—as proposed by Fuxman in [5]. SIA controls an explosion of the state transition space by mapping k-element means-end decompositions to k-conditional branch structures and k-element task decompositions to k-dimensional hypercube lattices. A detailed illustration of the algorithm and the significant improvement achieved w.r.t. the state transitional space complexity, has been documented in the original article [7]. This model transformation is achieved in the *i*ToNuSM V* tool by executing the *extract.exe* binary.

4.3.4 The Mapper Module

The mapper module takes the extended finite state model produced by the *Semantic Implosion Algorithm* and maps it to the input language of the NuSMV model verifier. We assign identifiers with all goals, tasks and resources that appear in the given SR-diagram. Each such identifier can have three possible values—`NC`, `CNF`, `FU`—corresponding to the three states mentioned in the previous section. All identifiers are initialized to the `NC` value which marks the initial state of our finite state model. The final state of the state model is denoted by any state where the root node has the value `FU`. The state transitions of the finite state model are captured using `next()` value assignments in the NuSMV input language. The *mapper.exe* binary does this mapping and generates an NuSMV input model as the final output.

4.3.5 i*ToNuSMV Output

The particular example shown in Fig. 4.16 generates a finite state model *STT.opm* and the corresponding NuSMV input model *NUSMV_input.smv*. *Var.opm* is a text file that contains the list of state variables that have been assigned to all the goals, tasks and resources. For the above example, *Var.opm* gets populated as shown below.

Also, according to the *Semantic Implosion Algorithm*, every state variable is initialized to zero, which represents the *Not Created* state of the entity. As the finite state model is built, every state variable makes two transitions. A $0 \rightarrow 1$ transition implies that the entity represented by that state variable goes from the *Not Created* state to the *Created Not Fulfilled* state. A $1 \rightarrow 2$ transition, on the other hand, implies that the represented entity goes from the *Created Not Fulfilled* state to the *Fulfilled* state. A sample finite state model for the healthcare example is shown in Table 4.7. The state variables used in the finite state model are in accordance with Table 4.6.

The same set of state variables are used to build the NuSMV input models. The NuSMV model corresponding to the finite state model shown in Table 4.7 is as follows:

```
MODULE main
VAR
V101 : NC, CNF, FU;
V102 : NC, CNF, FU; ....
```

All state variables are declared with enumerations NC (*Not Created*), CNF (*Created Not Fulfilled*), and FU (*Fulfilled*). Once all variables are declared, all the state variables are initialized to NC.

```
ASSIGN
init(V101) := NC;
init(V102) := NC; ....
```

After the declaration and initialization of state variables, we proceed to define the transition of state variables as captured in the finite state model of Table 4.7. For instance, rows 1, 6 and 15 of the finite state model represent transitions for state variable `V101`. We read these three lines of the finite state model and create the next state value for the *NuSMV* model as follows:

Table 4.6 State variable listing of entities

Variable name	Entity
V101	ProvideHealthcare
V102	Symptoms_Treat
V103	Symptoms_EMR_Treat
V105	ObtainSymptoms
V108	PatientEMR
V109	ConsultSpecialist

Table 4.7 Finite state model recorded in *STT.opm*

Present state	Event	Next state
V101 = 0	V101:0→1	V101 = 1,V102 = 0,V103 = 0
V101 = 1,V102 = 0,V103 = 0	V102:0→1	V101 = 1,V102 = 1,V103 = 0,V105 = 0
V101 = 1,V102 = 1,V103 = 0,V105 = 0	V105:0→1	V101 = 1,V102 = 1,V103 = 0,V105 = 1
V101 = 1,V102 = = 1,V103 = = 0, V105 = 1	V105:1→2	V101 = 1,V102 = 1,V103 = 0,V105 = 2
V101 = 1,V102 = 1,V103 = 0,V105 = 2	V102:1→2	V101 = 1,V102 = 2,V103 = 0
V101 = 1,V102 = 2,V103 = 0	V101:1→2	V101 = 2
V101 = 1,V102 = 0,V103 = 0	V103:0→1	V101 = 1,V102 = 0,V103 = 1,V105 = 0,V108 = 0,V109 = 0
V101 = 1,V102 = 0,V103 = 1,V105 = 0,V108 = 0,V109 = 0	V105:0→1	V101 = 1,V102 = 0,V103 = 1,V105 = 1,V108 = 0,V109 = 0
V101 = 1,V102 = 0,V103 = 1,V105 = 1,V108 = 0,V109 = 0	V105:1→2	V101 = 1,V102 = 0,V103 = 1,V105 = 2,V108 = 0,V109 = 0
V101 = 1,V102 = 0,V103 = 1,V105 = 2,V108 = 0,V109 = 0	V108:0→1	V101 = 1,V102 = 0,V103 = 1,V105 = 2,V108 = 1,V109 = 0
V101 = 1,V102 = 0,V103 = 1,V105 = 2,V108 = 1,V109 = 0	V108:1→2	V101 = 1,V102 = 0,V103 = 1,V105 = 2,V108 = 2,V109 = 0
V101 = 1,V102 = 0,V103 = 1,V105 = 2,V108 = 2,V109 = 0	V109:0→1	V101 = 1,V102 = 0,V103 = = 1,V105 = 2,V108 = 2,V109 = 1
V101 = 1,V102 = 0,V103 = 1,V105 = 2,V108 = 2,V109 = 1	V109:1→2	V101 = 1,V102 = 0,V103 = 1,V105 = 2,V108 = 2,V109 = 2
V101 = 1,V102 = 0,V103 = 1,V105 = 2,V108 = 2,V109 = 2	V103:1→2	V101 = 1,V102 = 0,V103 = 2
V101 = 1,V102 = 0,V103 = 2	V101:1→2	V101 = 2

```
next(V101) :=
case
V101=CNF & V102=NC & V103=FU : FU;
V101=CNF & V102=FU & V103=NC : FU;
V101=NC : CNF;
TRUE: V101;
esac;
```

This is done for all the state variables appearing in Table 4.6. The complete NuSMV model corresponding to the finite state model of Table 4.7 is obtained in *NUSMV_input.smv*.

4.3.6 The i*ToNuSMV Algorithm

An algorithm for the entire process may be specified as follows:
Input: Textual representation of an i* model SR-diagram.
Output: An extended finite state model and the corresponding NuSMV input model.

Algorithm:

Step-1: Tokenize the input text file using *filtokn.exe* and separate all tokens.
Step-2: Identify all the tokens and distinguish the keywords, user-defined variables and operators, separately, using *recognit.exe*.
Step-3: Identify the root model element from which the Semantic Implosion Algorithm will begin execution by using *modlroot.exe*. Associate state variables/identifiers with each goal, task and resource appearing in the i* model.
Step-4: Run the Semantic Implosion Algorithm by executing the *extract.exe* binary. This code generates the extended finite state model that can be derived from the i* model.
Step-5: Finally this finite state model is mapped to an NuSMV input model with the *mapper.exe* executable.
Step-6: *cleanup.exe* is used to clear the working directory before loading and converting the next i* model.

4.3.7 Platforms Used

The front end of the tool has been developed in the Microsoft Visual Basic environment. The 64-bit binaries have been generated using the Eclipse and Pelles C platforms.

4.3.8 Application Scenario

Let us consider the remote healthcare example illustrated in Fig. 4.16. A remote healthcare enterprise may want to comply to a temporal constraint that a *Doctor* will provide long term treatment only after it receives a *Symptoms Message* from the patient through the *ObtainSymptoms* goal. This implies that in the task decomposition of *Symptoms_EMR_Treat*, *ObtainSymptoms* must be Fulfilled before resource *PatientEMR* is acquired and task *ConsultSpecialist* is performed. This can be captured as a system property specified in CTL. The state variables listed in Table 4.6 can be used to define this property as follows:

$$AG((V103 = CNF \wedge V105 = FU \wedge \neg(V108 = FU \vee V109 = FU)) \rightarrow F(V105 = FU \wedge V108 = FU \wedge V109 = FU))$$

This property can be fed into the *NuSMV* model checker and verified against the *NUSMV_input.smv* input model generated by the *i*ToNuSM V* prototype. The NuSMV input model passes the model verification test if and only if all execution paths in the corresponding finite state model satisfy the condition that *V*105 is fulfilled before *V*108 and *V*109 are fulfilled. Otherwise, the CTL property is violated and *NuSMV* generates counterexamples.

4.4 i*ToNuSMV Version Manager

The tool has evolved through several versions as described below:

- **i*ToNuSMV Version 1.01**: Beta prototype that supported only 3-level goal models. Multi-actor scenarios were also not supported.
- **i*ToNuSMV Version 1.02**: This version accepts an i* model in non-standardized textual format and only converts it to the corresponding finite state machine and NuSMV input model. Model checking is not supported.
- **i*ToNuSMV Version 1.03**: The NuSMV model verifier is integrated into the tool. Users can now verify CTL specifications on the NuSMV model being generated.
- **i*ToNuSMV Version 1.04**: Bug fix. Previous version was path dependent. User was compelled to instal the tool in path C:Äù\istarToNuSMV. Path dependency removed.
- **i*ToNuSMV Version 2.01**: MAJOR UPGRADE. The input language of the tool has been changed from the previous non-standardized textual input to `tGRL` [8]. Supports multi-actor scenarios nut not inter-actor dependencies.
- **i*ToNuSMV Version 2.02:**: MAJOR UPGRADE. Inter-actor dependencies have been implemented.One finite state machine for the entire goal model is generated rather than peractor finite state machines as in the previous versions.

4.5 Contact and URL

The *i*ToNuSM V* tool can be freely downloaded from the following URL—http://cucse.org/faculty/tools/. The URL also contains a user manual of the *i*ToNuSM V* prototype in pdf format and can be downloaded from the link provided at the end of the page. For any further queries, please mail the authors at *novarun.db@gmail.com*.

4.6 Conclusion

Model checking tools, typically check a model against certain temporal properties. The need to bridge the gap between i* models and any other model with partial ordering is evident. Although model transformations have existed in the industry for quite some time, no work has been done to derive finite state models from i* models. This paper first illustrates and presents a *Naïve Algorithm* for extracting sequences from i* model constructs. Simulation results demonstrate how this causes a *hyperexponential explosion* in the finite state model space. The *Semantic Implosion Algorithm* provides an improvement to counter this explosion.

Detailed simulations have been done by applying both the algorithms to similar types of i* models and the results show that the *Semantic Implosion Algorithm* provides a significant improvement over the *Naïve Algorithm*. Typically, the finite state model space grows in the order of 10^{20} for the *Naïve Algorithm*, whereas, for the *Semantic Implosion Algorithm*, the growth rate is restricted to the order of 10^3. Although this may not be the best approach to extract a minimal set of plausible finite execution sequences, it definitely provides a significant improvement over the *Naïve Algorithm*.

The set of possible finite execution traces, that correspond to a given i* model, can be further pruned by feeding them into a model checking tool like NuSMV and checking them against certain enterprise-specific temporal properties or compliance rules. All models that generate counter-examples may be discarded. This is one of the biggest advantages of having a model that captures ordering of states. Also, once the set of valid finite state models have been obtained, we can map them to BPMN models, Petri Nets, or even UML models. This helps enterprise architects by allowing the automated generation of code snippets, thereby, reducing the efforts required to build the enterprise. Thus, once the requirements have been finalized and modelled by the architects, the development of the enterprise becomes fully automated. This ensures greater consistency and correctness and reduces the risks of failure.

The *i**ToNuSMV tool is a research prototype that takes a tGRL representation of an SR-model as input. The tool can also be extended to any goal modelling framework due to the generic nature of the model transforming Semantic Implosion Algorithm. A detailed working of the i*ToNuSMV tool with a multi-actor scenario having inter-actor dependencies is illustrated in the User Manual and Tutorial Video on the tool page.[1] Appropriate screenshots of the tool interface have also been provided.

[1] http://cucse.org/faculty/tools/.

References

1. Holzmann GJ (1997) The model checker SPIN. IEEE Trans Softw Eng 23(5):279–295. https://doi.org/10.1109/32.588521
2. Cimatti A, Clarke EM, Giunchiglia F, Roveri M (1999) NUSMV: a new symbolic model verifier. In: Proceedings of the 11th international conference on computer aided verification (CAV), pp 495–499. http://dl.acm.org/citation.cfm?id=647768.733923
3. Lapouchnian A (2005) Goal-oriented requirements engineering: an overview of the current research, Depth Report. University of Toronto. Canada, Toronto
4. Fuxman A, Pistore M, Mylopoulos J, Traverso P (2001) Model checking early requirements specifications in tropos. In: Proceedings of the 5th international symposium on requirements engineering (RE), pp 174–181. https://doi.org/10.1109/ISRE.2001.948557
5. Fuxman AD (2001) Formal analysis of early requirements specifications, MS thesis. Department of Computer Science. University of Toronto, Canada
6. Horkoff J, Yu E (2011) Analyzing goal models—different approaches and how to choose among them. In: Proceedings of the 2011 ACM symposium on applied computing (SAC), pp 75–682. https://doi.org/10.1145/1982185.1982334
7. Deb N, Chaki N, Ghose A (2016) Extracting finite state models from i* models. J Syst Softw, SI: COMPSAC, Elsevier 121:265–280. https://doi.org/10.1016/j.jss.2016.03.038
8. Abdelzad V, Amyot D, Alwidian SA, Lethbridge T (978) A textual syntax with tool support for the goal-oriented requirement language. In: iStar, vol 978, pp 61–66. http://ceur-ws.org/Vol-1402/paper6.pdf