An Approach to Generalising the State Space of a Distributed Missile Simulator

Steven Gordon, Lars Kristensen and Jonathan Billington
Computer Systems Engineering Centre
School of Electrical and Information Engineering
University of South Australia
Mawson Lakes SA 5095 Australia

Abstract. Formal methods can be used to verify refinements made in the design stages of systems. For example, the state space of a detailed design model can be compared with that of an abstract design model to see if it preserves sequences of events. Problems with state space analysis (state explosion, fixed initial states) make this difficult for real applications. In this paper we outline an approach for obtaining a generalised state space of a distributed missile simulator. The original state space has a repetitive structure. Our aim is to prove that for any initial state, the system will eventually halt, after which we can define a compact graphical representation of the state space, that is independent of the initial state.

INTRODUCTION

Background. Developing accurate requirements and design specifications is an important part of the systems engineering process. If errors are found in these specifications after the implementation phase, costly overruns in time and budget can occur. Modelling and analysis with formal methods can play a role in reducing errors by producing unambiguous specifications and also providing a means for proving properties of the system design. One property we are often interested in is whether a specification at one level of detail (e.g. an abstract design) is a faithful refinement of one at a higher level (e.g. requirements). State space analysis techniques (i.e. essentially exploring all possible states) can be used for proving such refinements (Valmari 1998). However, state space analysis is limited by the state explosion problem (too many states to reasonable calculate for complex systems) and the fact that a state space must be generated for each different initial state of the system. Reduction techniques may be used to alleviate these problems in some cases.

In this paper we discuss ideas for generalising a state space of a real application, a distributed missile simulator called Integrated Weapons Simulator (IWS) (Collas 1997). Two existing methods do not seem to work for our application. We present a new approach that combines induction, data-independence and parameterization. The state space of the detailed design model is repetitive. This leads us to analyse the structure of a single component and then inductively reason about the complete state space. From this we can obtain a general state space, with extra annotations, that represents any number of repetitive components, and hence a range of different initial states.

The following provides background information on the missile simulator and then we outline the contents of the remainder of the paper.

Missile Simulator. IWS is a distributed air-to-air missile simulator with a graphical user interface (GUI). It was developed for, and in collaboration with DSTO, the research arm of Australia's Department of Defence. The aim of IWS is to provide an environment for DSTO to test various algorithms for the guidance and control of air-to-air missiles. Accurate simulations may require the complexity of these algorithms to be large. Therefore, in designing IWS, several important features were desired so that the system performs adequately: modularity, concurrent execution and remote execution of separate components of the simulation. To provide modularity, IWS is divided into five components:

1. GUI: accept inputs from, and presents the results of the simulation to, the user.
2. Target: model the target for the simulation.
3. Radar: simulate the physical radar sensor on the missile.
4. Infrared: simulate the physical infrared sensor on the missile.
5. Missile Control: fusion of data from missile sensors, and guide and control the missile.

The four simulation components (Target, Radar, Infrared and Missile Control) can be executed remotely, and where possible, concurrently. In providing these features it is necessary to verify that communication between the components in IWS is correct. Coloured Petri nets (Jensen 1997) (CPNs) are used to model and analyse the communication in IWS.

Contents. The following Section describes the methodology that has been used for modelling and analysing IWS. A brief overview of CPNs is also given. Then we present the state space of the IWS detailed design model. Two existing ideas for generalising the state space, equivalence classes and

parameterized state spaces, are then given, and their pitfalls for our application discussed. Following this we present an outline of our new approach. We aim to prove the detailed design model eventually halts, and then we can graphically represent any configuration in a single general state space. We conclude this paper with a summary and future work.

It should be noted that, due to space limitations, we do not go into many more details about IWS and the CPN models. The reader interested in a detailed description of IWS is referred to (Gordon 1998).

MODELLING & ANALYSIS APPROACH

Coloured Petri Nets. CPN (Jensen 1997) is a formal method suited to expressing concurrency, non-determinism and system concepts at different levels of abstraction. They are a class of high-level nets that extend the features of basic Petri nets. We illustrate the important concepts of CPNs by using the requirements model of IWS, shown in Figure 1. The net consists of two types of nodes, *places* (ellipses, e.g. GUIState, Inputs and Outputs) and *transitions* (rectangles, e.g. Start, UpdateGUI, etc.), and directional arcs between nodes. An input arc goes from a place to a transition and an output arc vice versa. Places are typed by a *colour* set. For example, place GUIState has the colour set State. The declarations for Figure 1 are:

```
color State = with Start | Update | Compare | Halt;

color Inputs = with Delta;

color Outputs = with TargetPos | MissilePos | RadarRange |
InfraredRange;
```

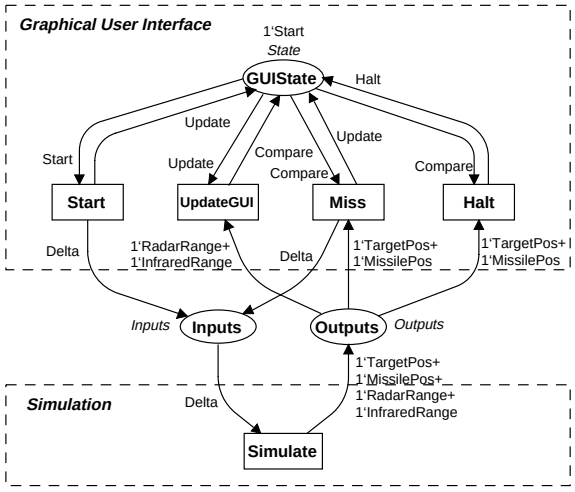
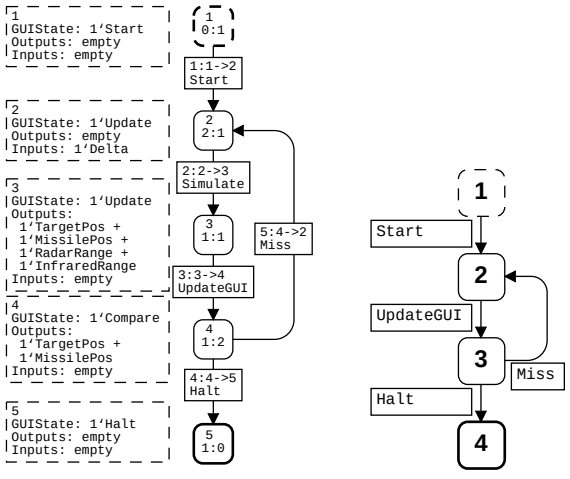


Figure 1: IWS Requirements Model

Places may be marked by a value from the colour set. These are known as *tokens*. For example, place GUIState is initially marked with one Start token, denoted by 1'Start above the place. A collection of tokens on a place is called it's *marking*, and the marking of the CPN comprises the markings of all places. Transitions and arcs also have inscriptions, which are expressions that, along with the tokens in places, determine whether a transition is *enabled*. A transition is enabled in a marking if sufficient tokens exist in each of its input places (as determined by the input arc inscriptions), and the transition *guard* (given in square brackets) evaluates to true.

In Figure 1, transition Start is enabled because Start is in the only input place and it is also the arc inscription, and there is no guard for the transition (which implies the guard is always true). A subset of the enabled transitions can *occur*. The occurrence of a transition destroys the necessary tokens in the input places and creates new tokens in the output places, as given by the expressions on the arcs. The occurrence of transition Start replaces the token Start with the token Update in place GUIState and creates a Delta token in place Inputs. When variables are used in arc inscriptions or guards, the values they are bound to on occurrence of a transition, together with the transition name, determine a *binding element*.

The CPNs of IWS were edited, simulated and partly analysed using Design/CPN. Design/CPN may be used to interactively or automatically simulate the net, and to create the *state space*. The state space is a directed graph with nodes and arcs representing the net markings and binding elements, respectively (e.g. Figure 2(a)). A complete state space represents all possible states the CPN can reach. In Design/CPN queries can be made on the state space to determine dynamic properties of the CPN (e.g. deadlocks, live-locks, bounds on places). The state space can also be viewed as a finite state automaton (FSA) which, with appropriate analysis techniques can be used to give the language accepted by the CPN, where the binding elements are the alphabet (e.g. Figure 2(b)). Language comparison of different models is used to prove the faithful refinement, in terms of sequences of certain events, of the models.



(a) (b)

Figure 2: Requirements (a) State Space & (b) FSA

Methodology. IWS has been modelled and analysed at three levels of abstraction: requirements, abstract design and detailed design. A top-down methodology,

as seen in Figure 3, has been followed.

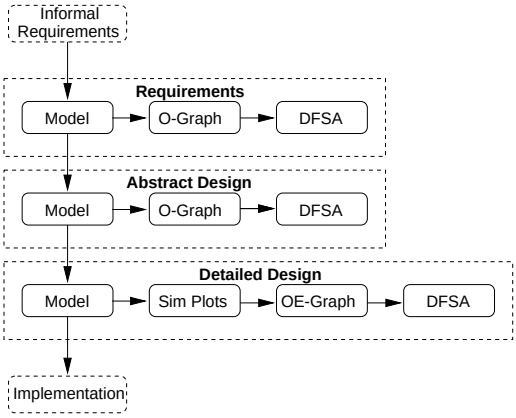


Figure 3: Modelling & Analysis Methodology

The requirements shows the sequence of events that define the interactions between the GUI and the simulator. The events are given by the key transitions Start, UpdateGUI, Miss and Halt. The deterministic FSA (DFSA) of the requirements model (Figure 2(b)) is used when determining whether the design models satisfy the requirements. That is, for the abstract design to be a faithful refinement of the requirements, their DFSAs must be the same.

The CPN of the abstract design models the message flow between the GUI and simulator and, more importantly, within the simulator. The model consists of a top-level page which models the GUI-simulator interactions and the communication between simulator components, and sub-pages describing the message flow within each of the four simulator components. The basic flow of data (see Figure 4 for the state space) is as follows: the target trajectory is updated based on the target model (starting at node 2); radar and infrared ranges are updated based on the target and missile trajectories; and the missile trajctory is updated based on the estimated ranges and current missile trajectory. This flow is one iteration. At the end (i.e. node 31) a test is made if a hit occurs (i.e. the missile collides with the target) or the user manually stops the simulator. If so, the simulator is halted (node 32), otherwise a miss occurs and another iteration begins (node 2).

It should be noted that the abstract design uses constant values to represent the data in the simualtor (e.g. target and missile trajectories, radar and infrared ranges). The next model, the detailed design (Figure 5), has the same CPN structure as the abstract design, but introduces actual data values such as the x,y,z coordinates and fucntions to manipulate them. Therefore, as well as modelling the flow of data, the detailed design also models the calculations performed within the simulator. For more details on the modelling and analysis see (Gordon 1998). Table 1 summarizes the information included in each of the models. One aim of the analysis is to ensure each model is a faithful refinement of its predecessor.

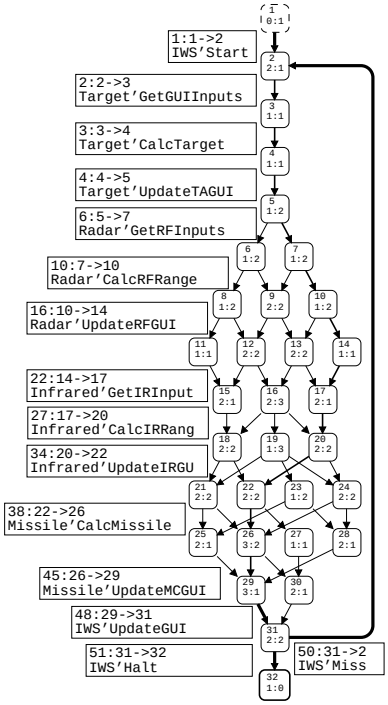


Figure 4: Abstract Design State Space

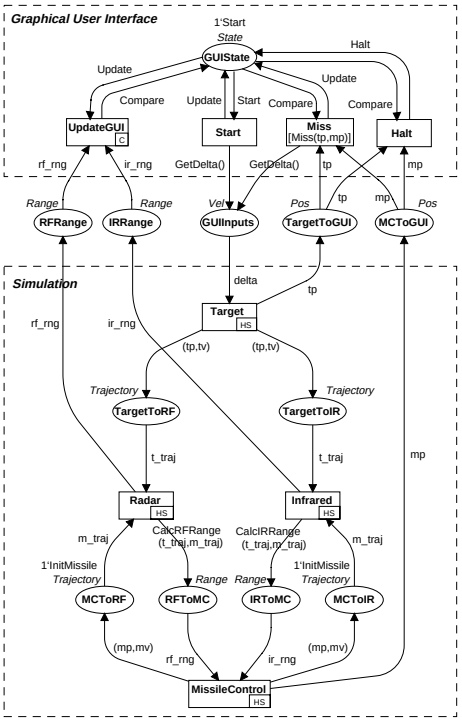


Figure 5: Detailed Design CPN Model

Model	GUI to Simulator	Simulator Data Flow	Simulator Calculations
Requirements	*		
Abstract Design	*	*	
Detailed Design	*	*	*

Table 1: Summary of Modelled Features

STATE SPACE ANALYSIS OF IWS

Ordinary State Space. The state space of the detailed design model depends on the initial parameters used in the model. These parameters include the initial missile and target positions, the maximum missile and target speeds and the tolerance for a hit to occur (i.e. the distance between the target and missile to be classified as a hit). When the target is initially further away from the missile, then the simulator will take more iterations before a hit occurs than if the target was closer. Table 2 shows some state space results for different configurations.

Parameter Values			State Space Results		
Tol.	Target Pos.	Target Vel.	Nodes	Arcs	TS
0.01	(0.025,0,0)	$(10^3,0,0)$	94	150	3
0.01	(1,0,0)	$(10^3, 10^3, 10^3)$	7410	11950	239
0.005	(1,0,0)	$(10^3, 10^3, 10^3)$	14509	23400	468
0.01	(1,0,0)	$(10^3,0,0)$	5457	8800	176

Table 2: Detailed Design State Space Results. Tol = tolerance, TS = terminal states, Units: km and hours.

(Gordon 1998) give an interpretation of the results in Table 2. For simplicity and brevity, we will use the state space from the first row of Table 2, shown in Figure 6, to explain the dynamics of the detailed design model.

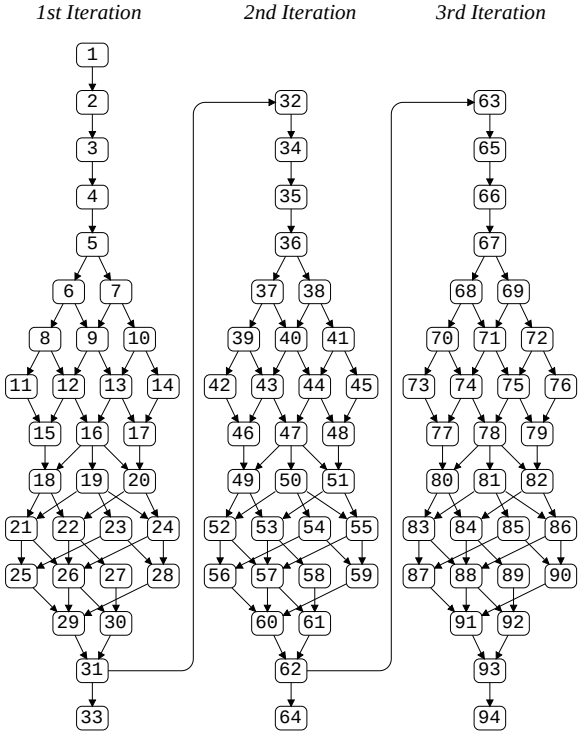


Figure 6: Detailed Design State Space

We have presented Figure 6 so we can see the correspondence of the state space to the iterations in the simulator. For the initial three iterations occur before the missile hits the target. After IWS is started (the Start transition occurs) a simulation iteration begins. We can classify the nodes 2 through to 31 as performing the calculations in the simulator. At node 31, a user stop can occur, leading to a terminal state, node 33. Alternatively, the Miss transition can occur as the target and missile positions are not close enough to classify as a hit. This leads to node 32 which begins a second iteration. Again, the calculations are performed (nodes 32 to 62) and either a user stop or miss can occur. Similarly, the third iteration occurs until we get to node 93. At this point the missile and target positions are close enough to classify as a hit. Therefore Miss cannot occur, and the only option is for Halt to occur, leading to the final terminal state, node 94.

For the trivial example we have chosen we can visually examine the state space. However, in general, two problems arise when performing state space analysis of the detailed design model: (1) for some initial states the state space will be too large to calculate, and (2) it is impossible to investigate every scenario, i.e. all initial states.

Therefore, it is desirable to obtain a general state space that is representative of the dynamics of the detailed design model under all reasonable configurations.

GENERALISING THE STATE SPACE

Intuition. Our thoughts on obtaining a general state space for the detailed design have been based on the fact that the state spaces have a repetitive structure. For each iteration there is a sub-graph, the structure of which is the same for the different iterations, but the data within the sub-graph of one iteration is different from the data in another iteration. However, looking more closely at the data, we see that it is related. For example, in Figure 6, the target trajectory in node 32 (iteration 2) is related (by a function) to the target trajectory in node 2 (iteration 1). To obtain a generalised state space we need to: relate the data across iterations, and merge nodes which are only differentiated by the data.

Figure 6 may be viewed as an unwinding of Figure 4 for three iterations. Intuitively, we would like to perform the reverse operation and show that Figure 6 can be merged to obtain Figure 4. That is, we would like to merge iteration 3 into iteration 2, which in turn is merged into iteration 1, so we obtain a graph with 32 nodes, with node 31 having an arc back to node 2. This would have an identical structure to the abstract design state space (Figure 4), although different annotations. As we will see later, there are boundary conditions that also need special consideration. The following gives an overview of two possible approaches of obtaining a generalised state space.

Equivalence Classes. State spaces with equivalence classes is a state space reduction technique defined in (Jensen 1997b) for Coloured Petri nets that also have

tool support in Design/CPN. To use equivalence classes, two equivalence relations need to be provided: one on the markings and the other on the binding elements. Each relation takes two markings/binding elements as inputs and returns true if the markings/binding elements are equivalent. In our case, for example, the equivalence relation for the markings would define equivalence when the markings for each individual place are the same, except for data values, when it is equivalent if each data value is a function of the other data value (e.g. target trajectory in iteration 2 = F(target trajectory in iteration 1)).

All markings that are equivalent are placed in an equivalence class. Similarly, for binding elements. A state space with equivalence classes draws equivalence classes of marking/binding elements as nodes/arcs. At first thought, applying equivalence classes to the detailed design model would provide us with a general state space. The graph would be similar to the abstract design state space (Figure 4) where, for example, node 2 represents the class of markings including nodes 2, 32 and 63 in Figure 6. However, to be able to relate the state space with equivalence classes back to the ordinary state space (which is required for the properties of state spaces to be maintained e.g. deadlocks), requires certain conditions on the equivalence relations.

Assuming Figure 4 is our state space with equivalence classes, the arc from node 31 to 2 in Figure 4 essentially says we can always start a new iteration. However, in the detailed design state space (Figure 6) once the missile is within tolerance for hitting the target (e.g. node 93) we cannot start a new iteration, we can only halt as the target has been hit. Therefore our state space with equivalence classes is incorrect. This is due to the equivalence relations being inconsistent. A definition of consistency is given in (Jensen 1997b).

To compare the abstract and detailed designs, we have to take into account a subtle difference: the abstract design allows for infinitely many iterations while the detailed design, for any reasonable configuration, results in a finite number of iterations. We have introduced a condition into the detailed design model that implies the simulator must eventually halt. This is based on an assumption that we must make regarding the behaviour of the missile and target, and that is that the missile is smart enough to track down the target. This may not be true in general, but it is a scenario which we would like to investigate. Our generalised state space must represent this eventual halting property. The simplest way seems to annotate the arc from node 31 to 2 in Figure 4 with a condition of the form 'this arc exists for all iterations except the last one'.

Parameterized State Spaces. Parameterized state spaces allow us to represent state spaces in a symbolic way. They are defined for Predicate/Transition (Lindqvist 1993) and Algebraic Petri nets (Schmidt 1995) (two other classes of high-level nets) but there is no current extension for Coloured Petri nets. Parameters can be used in the markings to represent multiple, similar markings. A simple example follows in Figure 7.

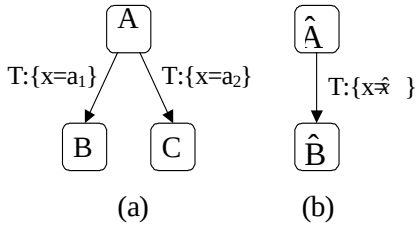


Figure 7: Example of (a) an ordinary & (b) a parameterized state space

Rather than using the explicit values, a_1 and a_2, in the ordinary state space, we can represent them by a parameter, $\hat{x}$.

If we consider another example where we have two similar parameterized markings (e.g. the number of tokens in each place is the same), A and B, (or binding elements), in some cases we can say marking A includes marking B. If this is the case there is no need to generate the successor markings from B – we already have them from A. We could apply this to the detailed design model. In Figure 6, if node 2 and node 32 were parameterized in a way that 32 was included in 2, we could merge them. Again we would have an arc going from node 31 to 2. This arc would need a condition stating the binding element exists only when the missile has not hit the target (which is possible using parameters).

Parameterized state spaces may give us a formal way of representing symbolic information in our state space (e.g. the condition on the Miss arc saying it doesn't exist for the last iteration) but there are several questions raised:

1. How are they defined for CPNs?
2. How do we parameterize the markings to obtain a reduced state space? For example, to merge nodes, the parameterized markings 2, 32 and 63 in Figure 6 must have the same number of tokens in each place and the marking of each place must be a similar parameterized expression. If place GUIInputs is marked with delta1 in node 2 and delta2 in node 32 then the two parameters delta1 and delta2 must include the same possible range of explicit values.
3. Does the consistency problem still exist when we have a condition on the arc?

At this stage, parameterized state spaces do not seem feasible to use to obtain a generalised state space of the detailed design model.

A NEW APPROACH

Overview. The consistency problem discovered using equivalence classes must be addressed in a generalised state space. That is, the generalised state space must reflect the assumption that the system must halt, as opposed to the abstract design where, at a higher level of abstraction, the number of iterations is unbounded. We aim at proving that for any initial state the detailed design eventually halts. Our proposed approach is this:

1. Show that essentially our state space is composed of iterations, and we can examine the structure of a general iteration, and say that it applies to all others.
2. Prove that if we input some K into the general iteration we output some K-1. Here K is a metric related to the closeness of the target and missile, and K-1 is a state closer to the end of the simulation than K. First we must show that we can test a condition (e.g. the distance between target and missile) to determine if we have reached the halt state. Let that distance be defined as K=0. If we now compose the general iterations, and input a fixed value for K, we will eventually reach the end of the simulation, or the halt state (0).
3. Given that we know the system will eventually halt, and we have a general iteration, we can graphically represent all configurations of the detailed design model composed of general iterations, as a generalised state space.

The following subsections explain these steps in more detail, using the example shown in Figure 8, similar to our detailed design state space.

Structure of the CPN Model. We have already noted the detailed design state space is structured into sub-graphs that represent iterations in the simulation. The CPN model can be used to prove that the state space is structured in this way. In Figure 5 a simulation iteration begins when transition Start occurs. The next and all following iterations begin when transition Miss occurs. We can think of our CPN model as a repetition of the transitions and places for one iteration. The execution of these iterations can easily be related to the iterations in our state space. We can see that, as along as we carefully consider the boundary conditions (which we will later), it is satisfactory to examine a general iteration, and extend our results for any number of composed iterations.

Examining a Single Iteration. An iteration begins when the transition Start or Miss occurs, and ends after all calculations have been performed and the GUI is updated. To prove that the system eventually halts, we need to prove that after each iteration we get closer to the hit state (i.e. after a Hit occurs). A metric must be defined for an iteration that tells us if we are getting closer to the hit state. In general, we desire a

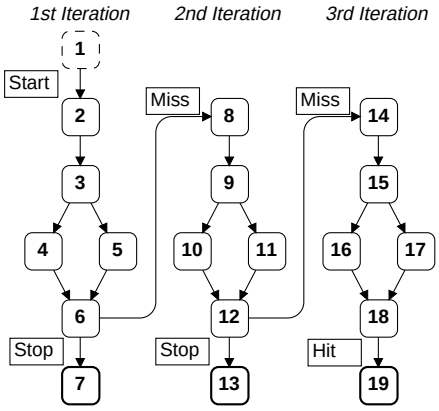


Figure 8: Example State Space

structure similar to Figure 9. Here, K is the metric, and we define the hit state as 0. When K is input at node 2, K-1 is output at node 6. If we take initial K = 3, for example, after performing the iteration the output will be 2. By composing 3 iterations, if we input K = 3, we will output 0. This means we have reached the halt state. Unfortunately, the missile simulator is not as simple as decrementing one value during the iteration. In the following we describe the important issues for examining a single iteration.

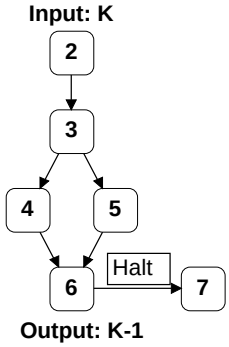


Figure 9: Structure of Single, General Iteration

The distance between the missile and target, *d*, will be used as the metric for our analysis. Although we don't explicitly use this value in the detailed design CPN, we can derive it from the missile and target positions. When this distance is less than or equal to the tolerance then we know a hit will occur, this is how we define the hit state. Note that the granularity of the simulation is chosen such that the missile cannot pass the target in a single iteration without a hit being identified.

During an iteration, the missile and target positions (and hence, *d*) are manipulated. If we keep track of all the changes to *d* that occur during the iteration we will output a term $d - a$, where a is another term. a may comprise functions and values. If we show that a is always positive, then we know that given an input d to the iteration, we will output $d - a$ which is less than d. Hence, the distance between missile and target always decreases after each iteration. Since we are using reals for *d*, we must specify an adequate

quantum for measurements, e.g. we can only measure distances to the nearest metre. If so, it follows that by composing any number of iterations, we will eventually reach the hit state, i.e. when *d* is less than the tolerance.

Of course, our approach depends on *a* being positive. Although we believe our functions used in the model will produce this, we will not know for sure until further analysis is completed (this is discussed at the end of this Section). One calculation that needs special consideration is the updating of the target trajectory. Until now, this has been by adding a *delta* value, obtained by the function GetDelta(), during every iteration. GetDelta() aims to represent the changes to the target by the user. We have done very little investigation into what type of function is best suited for GetDelta(). But to ensure our system eventually halts, we now have to pay attention to this function. It must be chosen in such a way that the distance between the target and missile converges. Eventually, we may be able to define a class of functions for which this occurs. Then we are guaranteed that the missile will get close enough to the target for a hit to occur.

Determining the term *a* can be performed using our CPN model (Figure 5). This is advantageous because the majority of our proof can then be done with the computer tool Design/CPN. We can take the CPN structure that represents the simulation iteration and change the declarations and inscriptions to symbolic values in terms of the metric we are using. For example, when we add delta to the target trajectory, we also add it to *d*. Rather than performing the calculation on the values (e.g. (100,50,50) + (3,6,5)) in the CPN we would operate on the term (e.g. *d* + *delta*). The state space for this CPN would give us the term *a* that is output. The details of the implementation are out of the scope of this paper, and still need further investigation.

GRAPHICAL REPRESENTATION

A Compact Representation. The limitation of equivalence classes and parameterized state spaces was the representation of the final iteration – they didn't necessarily convey the information that the system will eventually halt. By following the approach of the preceding Section, we can prove that eventual halting is guaranteed. The state space for any configuration of the detailed design can be drawn as a composition of general iterations. With this knowledge, we can use a compact graphical state space to represent the state space of the detailed design in any configuration. Figure 10 is our generalised state space.

Defining the Representation. It is not enough to simply draw our general state space as in Figure 10. In fact it is not a state space in the strict sense – they have no notion of symbolic values, just fixed values.

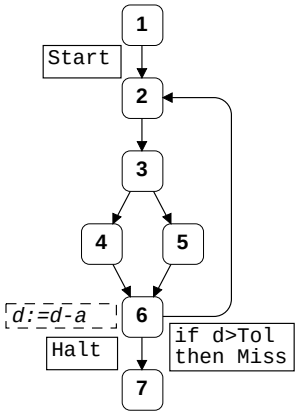


Figure 10: Generalised State Space

We need to define what our generalised state space (which we will continue to call it) represents.

The structure of Figure 10 is similar to any normal state space where we have a directed graph with nodes representing markings and arcs representing binding elements. We have introduced additional information to represent the metric being used. In Figure 10, the dashed box shows that at the end of an iteration (node 6), the metric has been updated, i.e. *d*:= *d*-*a*. Another modification is the binding element for the transition Miss is conditional. This means Miss can occur only if the distance between missile and target is greater than the tolerance. If not, this arc does not exist. When unwinding the generalised state space we generate new nodes when we start each new iteration. From this we can obtain the structure of the detailed design state space.

To include specific data in the unwound state space we must instantiate the markings and binding elements for some given initial state. To start a new iteration we need to update the values in the first node, based on the values used at the end of the previous iteration. It is necessary to provide a formal definition of this generalised state space, and relate them to the CPN model or ordinary state space. This is left as future work.

DISCUSSION

Summary. We have sketched an approach for obtaining a general state space for a detailed design model of a distributed missile simulator. Its ordinary state space comprised a number of iterations, each with the same structure. This was problematic as we may have a large number of iterations, and each different initial configuration of the system produces a different state space. With the repetitive structure, it seems intuitively obvious that we should be able to represent all the information in a single general state space.

Equivalence classes with state spaces and parameterized state spaces are two approaches for reducing state spaces that seemed applicable to the missile simulator problem. However, neither seem to

represent the fact that the system will eventually halt. Our approach is to show from our model that we have a general iterative structure, and in any configuration, by composing the iterations we will get the ordinary state space. We use this to also prove that the system will eventually halt. A compact graphical representation of the state space can then be given that includes all the necessary information.

It is important to note the differences in the approaches to solving the problem. Equivalence classes and parameterized state spaces involve applying a known formal method to the problem. Their application seemed feasible because initial thoughts suggested they would give state spaces similar to the abstract design. However, they revealed that an important property in obtaining a general detailed design state space was to show it eventually halted. Our approach, starts with proving the eventual halting property, and then it follows that with some extra annotations on the state space, we can obtain a generalised state space.

The general state space can be used to see how the detailed design differs from the abstract design using automata reduction and comparison techniques. In turn, the two design models can be compared to the requirements model, to determine if they are all faithful refinements.

Future Work. We have only presented the approach for obtaining a generalised state space of the detailed design model. The complete application of this approach to the distributed missile simulator needs to be completed. Some important issues that need resolution are:

- Defining the class of appropriate functions for GetDelta() that result in a convergence of the missile and target positions.
- Defining our metric, e.g. where and how do we keep track of the current iteration.
- Using our CPN model and Design/CPN to show that our metric input into a iteration will be output with a value closer to the halt state.

Although we are confident with our graphical representation, for completeness we intend to provide a formal defintion of it, in relation to the CPN model.

Finally, our approach has been specifically for our application, the distributed missile simulator. We need to investigate to what extent it can be generalized and the types of applications that it may be applied to.

REFERENCES

Collas, B., Gordon, S., and Widjaja, H., *IWS Design Specification V1.1*. Uni. of SA, Adelaide, 1997.

Gordon, S. and Billington, J., "Analysing a missile simulator using Coloured Petri nets." *Int'l Journal on Software Tools for Technology Transfer*, Vol. 2, No. 2, pp. 144-159, Dec. 1998.

Jensen, K., *Coloured Petri Nets. Basic Concepts, Analysis Method and Practical Use*. Volumes 1-3. Springer-Verlag, Berlin, 1997.

Lindqvist, M., "Parameterized reachability trees for Predicate/Transition nets." *Advances in Petri Nets.* LNCS 674. Springer-Verlag, Berlin Heidelberg New York, 1993, pp. 301-324.

Meta Software Corporation, *Design/CPN Reference Manual for X-Windows*. Version 2.0, Meta Software Corporation, Cambridge, MA, 1993.

Schmidt, K., "Parameterized reachability trees for Algebraic Petri nets." *Proc. 16th Int'l Conf. Application and Theory of Petri*. LNCS 935. Springer-Verlag, Berlin, 1995, pp. 392-411.

Valmari, A., "The state explosion problem." *Lectures on Petri Nets: Advances in Petri Nets Volume II.* Springer-Verlag, Berlin, 1998, pp 492-528.

BIOGRAPHIES

Steven Gordon completed a BEng in Computer Systems Engineering at UniSA in 1997, receiving prizes in a number of subjects. Since then he has been a PhD candidate within the Institute for Telecommunications Research and the Computer Systems Engineering Centre. His research has been on the general area of verification of distributed systems using Coloured Petri nets, which has been applied to a distributed missile simulator and the WAP Wireless Transaction Protocol. He has had brief stints working on ITR/CSEC projects, with DSTO and Telstra in similar areas.

Lars Michael Kristensen received a Ph.D in computer science from the University of Aarhus, Denmark in 2000. He received a Master of Science degree in computer science and mathematics from the University of Aarhus in 1997. He has been working as a Research Assistant Professor at the University of Aarhus, and is currently working as a Research Associate at the Computer Systems Engineering Centre at the Univesity of South Australia, Adelaide. His main research interest include Formal Methods, Coloured Petri Nets, State Space Methods, Communication Protocols and Data Networks, Concurrent Systems, Computer Tools for Validation and Verification.

Jonathan Billington has B.E. and MEngSc degrees from Monash University, Australia and a PhD from the University of Cambridge,UK. After working for Adelaide University, he spent 15 years with Telecom Australia Research Laboratories, where he led a team developing protocol engineering tools and techniques. Jonathan is Professor of Computer Systems Engineering at the University of South Australia and the Director of the Computer Systems Engineering Centre, where he leads a group researching distributed and concurrent systems. He has consulted to various companies and government agencies and is currently editor of ISO/IEC 15909 on High-level Petri nets.