Controlling Modelling Artifacts

Michael J.A. Smith, Flemming Nielson and Hanne Riis Nielson
Department of Informatics and Mathematical Modelling
Danmarks Tekniske Universitet
Lyngby, Denmark
Email: {mjas, nielson, riis}@imm.dtu.dk

***Abstract*—When analysing the performance of a complex system, we typically build *abstract* models that are small enough to analyse, but still capture the relevant details of the system. But it is difficult to know whether the model accurately describes the real system, or if its behaviour is due to *modelling artifacts* that were inadvertently introduced.**

In this paper, we propose a novel methodology to reason about modelling artifacts, given a detailed model and a high-level (more abstract) model of the same system. By a series of automated abstraction steps, we lift the detailed model to the same state space as the high-level model, so that they can be directly compared. There are two key ideas in our approach — a *temporal abstraction*, where we only look at the state of the system at certain observable points in time, and a *spatial abstraction*, where we project onto a smaller state space that summarises the possible configurations of the system (for example, by counting the number of components in a certain state).

We motivate our methodology with a case study of the LMAC protocol for wireless sensor networks. In particular, we investigate the accuracy of a recently proposed high-level model of LMAC, and identify some modelling artifacts in the model. Since we can apply our abstractions on-the-fly, while exploring the state space of the detailed model, we can analyse larger networks than are possible with existing techniques.

I. Introduction

Modelling is a tricky business. We build models because we want to analyse systems that are too complicated to look at directly, so that we can subject them to scenarios that might be difficult to test in practice, and so that we can gain a greater understanding of how they behave. It is precisely because these systems are so complicated that we need to choose a level of *abstraction* when modelling them — deciding which details are relevant, and which we can ignore without having a large impact on the results.

The problem here is that the more abstract our model is, the more likely we are to introduce *modelling artifacts*. That is to say, we make assumptions in the model that are not necessarily true of the real system, and so the behaviour of the model diverges from that of the real system. We make a trade-off, in the sense that more abstract models are smaller and easier to analyse, but may be less accurate than more detailed models.

The topic of this paper is concerned with a novel methodology to help us avoid introducing modelling artifacts, while still gaining from the benefits of more abstract models — namely, that we can analyse larger systems. Our approach is to start with a detailed model, and to perform a series of abstraction steps that allows us to build a more abstract model. The novelty comes, in particular, from the way we *combine* existing abstraction techniques to yield a powerful new abstraction framework. This is an important contribution in two ways:

1) If we have a detailed model and an abstract model of the same system, it allows us to *compare them*, so that we can reason about the modelling artifacts that were introduced by the assumptions in the abstract model. This is not to say that the detailed model is itself free of modelling artifacts, but since it is closer to the actual system, there are fewer opportunities for modelling artifacts to arise.
2) Our methodology allows us to *automatically build an abstract model* from a detailed model, so that we can analyse larger systems than are possible with existing techniques, without introducing any additional modelling artifacts in the abstract model. Furthermore, we have built an efficient implementation by performing the abstraction *on-the-fly*.

The models we consider in this paper are *probabilistic*, and so the questions we want to answer are, for example, of the form "what is the probability that the system is in a state with a certain property, after it has been running for a certain time?"

We begin the paper in Section II with a description of probabilistic modelling techniques, using Discrete Time Markov Chains (DTMCs), and higher-level modelling languages built on top of this formalism. In particular, we will look at the different approaches to building high-level (abstract) models and low-level (detailed) models, using the Lightweight Medium Access Control (LMAC) protocol for wireless sensor networks [15] as a running example.

We present our methodology in two stages. The first, which we call *temporal abstraction*, is based on the idea that we can only observe the state of the model *at certain points in time*, and we describe this in Section III. In general, since we consider time-abstract models, these 'points in time' are defined in relation to one of the components in the model — we only observe the state of the model when this component changes its state. If we call this component a 'clock' (in an abstract sense), this means that we can only observe the state of the model when the clock 'ticks' (i.e. there is some global agreement of time passing). The intermediate states between these events are hidden from us. Importantly, we can shift our temporal granularity by only *sampling* the state of the model every t 'ticks'.

The second stage in our methodology, described in Section IV, is called *spatial abstraction*. The idea is that we are only interested in certain derived properties of the model, such as the number of components in a particular state, rather than the particular state of every single component. We can therefore *summarise* the configurations of the model by projecting its states onto a more abstract state space. Because we want to apply this methodology to models without any obvious symmetry — such as a model of a network with a non-uniform topology — this abstraction will typically result in a Markov Decision Process (MDP) rather than a DTMC [6]. In other words, the abstraction introduces non-determinism, or uncertainty, but it allows us to *safely bound* properties of interest, such as the probability that a property holds within a certain number of time steps.

Before concluding the paper, we present some results of applying our methodology to the LMAC protocol, in Section V. This uses a tool that we have implemented, and allows us to compare a detailed model of the protocol with a very abstract model, based on combinatorial arguments, that was recently proposed in [7]. In particular, we can identify some modelling artifacts that were introduced in this model. Further to this, we present some performance results for our tool, illustrating that we are able to analyse much larger models than have previously been possible, by building the abstract model *on-the-fly* while exploring the state space of the detailed model.

II. High- and Low-Level Modelling

The basic mathematical model that we use in this paper is the Discrete Time Markov Chain, which is defined as:

Definition 1. *A* Discrete Time Markov Chain (DTMC) *is a tuple* $(S, \boldsymbol{P}, \iota)$*, where* S *is a finite non-empty set of states,* $\boldsymbol{P} : S \rightarrow S \rightarrow [0,1]$ *associates a probability distribution over the state space* S *to each* $s \in S$*, and* $\iota \in S$ *is the initial state. We require for all* $s \in S$ *that* $\sum_{s' \in S} \boldsymbol{P}(s)(s') = 1$.

We will consider DTMCs with a structured state space, such that $S \subseteq V_1 \times \cdots \times V_n$, where V_i are finite non-empty sets.

It is common to augment the above definition of a DTMC with a labelling function L, which assigns a set of atomic propositions to each state. This is so that we can refer to sets of states when we specify properties of the model that we would like to verify. An equivalent approach is to define each atomic proposition $a \in AP$ to be a function $a : V_1 \times \cdots \times V_n \rightarrow \mathbb{B}$, which evaluates to true or false for each state in the model.

A popular logic for expressing properties of DTMCs is Probabilistic Computation Tree Logic (PCTL) [10]. Since we do not need PCTL in its entirety to present the ideas in this paper, we will focus on just two important quantitative properties of DTMCs — bounded and unbounded reachability[1]. A bounded reachability property $\mathrm{Pr}_s(\lozenge^{\leq n} a)$ queries the probability that, starting in state s, we reach a state t within n steps such that $a(t)$ holds. We can compute this iteratively as follows, where $n \in \mathbb{N}_0$ is a non-negative integer.

$$\mathrm{Pr}_s(\lozenge^{\leq n} a) = \begin{cases} 0 & \text{if } n = 0 \wedge \neg a(s) \\ 1 & \text{if } a(s) \\ \displaystyle\sum_{t \in S} \boldsymbol{P}(s)(t) \cdot \mathrm{Pr}_t(\lozenge^{\leq n-1} a) & \text{otherwise} \end{cases}$$

An unbounded reachability property $\mathrm{Pr}_s(\lozenge a)$ relaxes the constraint on the number of steps. This can be computed by solving a set of linear equations, or (as is more commonly done in practice) by approximating the following limit:

$$\mathrm{Pr}_s(\lozenge a) = \lim_{n \rightarrow \infty} \mathrm{Pr}_s(\lozenge^{\leq n} a)$$

We will make use of this in Section III when we compute temporal abstractions.

When modelling a system using a DTMC, there are two different approaches we can take. One approach is to *directly write down* the state space S and probability transition matrix $\boldsymbol{P}$ of the DTMC. Since we clearly do not want to specify each state separately, we find a *parametric* way of describing the model. That is to say, we have a structured state space, and define the transitions from each state as a function of the state. For this to be possible, there must be a lot of symmetry in the DTMC we are describing, and so we typically view the system at a fairly high level of abstraction. This can lead to models that scale reasonably well, but we must be very careful about introducing modelling artifacts.

The second approach, more commonly used by computer scientists, is to specify the DTMC using a higher-level *modelling language*. Here, we model the system in a compositional way, describing the individual behaviour of each component and composing them to build a model of the entire system. We can automate the generation of a DTMC from the model, based on the semantics of the modelling language. This typically leads to more detailed models, which can become very large, but are less likely to contain modelling artifacts.

The techniques we describe in this paper can be applied to any compositional modelling language whose underlying semantics is a DTMC, and where there is a notion of synchronisation over actions. To apply our methodology to a real case study, however, we need to instantiate it on a particular language, and to this end we use the PRISM language [11]. A PRISM model consists of a number of *modules*, each containing a number of *variables*, which can either be Boolean, or take on integer values over a finite range. The evolution of each module is described by guarded commands of the form:

$$[a]\ G \rightarrow p_1 : U_1 + \cdots + p_n : U_n\ ;$$

Here, a is an *action type* associated with the command, G is a *guard* — an expression over the variables in the model — and U_i are *updates* — changing the values of certain variables to specify a new state. If G evaluates to true, then update U_i is applied with probability p_i (we require $\sum_i p_i = 1$).

Since there may be more than one command enabled at the same time (i.e. there are two guards that evaluate to true for the same state), the semantics of a PRISM module in general induces a *Markov Decision Process (MDP)*.

[1] These are technically not expressible in the logic of [10], but are implemented (e.g. in PRISM), by means of a quantitative ‘`P=?`’ operator.

Definition 2. *A* Markov Decision Process (MDP) *is a tuple* $(S, Act, \boldsymbol{P}, \iota)$*, where* S *is a finite non-empty set of states,* Act *is a finite non-empty set of actions,* $\boldsymbol{P} : S \to Act \to \mathcal{P}(S \to [0,1])$ *associates a finite set of distributions over the state space* S *to each state and action, and* $\iota \in S$ *is the initial state. We require for all* $s \in S$ *and* $a \in Act$ *that for all* $\boldsymbol{\pi} \in \boldsymbol{P}(s)(a)$*,* $\sum_{s' \in S} \boldsymbol{\pi}(s') = 1$*.*

If we tell PRISM that a model represents a DTMC, it resolves the non-determinism in the above by giving an equal probability of choosing between concurrently-enabled commands. It is useful for our purposes to *remember* the action labels when we do this (we will make use of this in Section III), and so we need to additionally define the notion of a *labelled DTMC*:

Definition 3. *A* labelled DTMC *is a tuple* $(S, Act, \boldsymbol{P}, \iota)$*, where* S *is a finite non-empty set of states,* Act *is a finite non-empty set of actions,* $\boldsymbol{P} : S \to Act \to S \to [0,1]$ *associates a probability distribution over* $Act \times S$ *to each* $s \in S$*, and* $\iota \in S$ *is the initial state. We require for all* $s \in S$ *that* $\sum_{a \in Act} \sum_{s' \in S} \boldsymbol{P}(s)(a)(s') = 1$*.*

When PRISM treats a model as a DTMC, it induces a labelled DTMC $\overline{M} = (S, Act, \overline{\boldsymbol{P}}, \iota)$ from an MDP $M = (S, Act, \boldsymbol{P}, \iota)$ as follows:

$$\overline{\boldsymbol{P}}(s)(a)(s') = \begin{cases} \dfrac{1}{|\boldsymbol{P}(s)|} \displaystyle\sum_{\boldsymbol{\pi} \in \boldsymbol{P}(s)(a)} \boldsymbol{\pi}(s') & \text{if } |\boldsymbol{P}(s)| > 0 \\ \mathbf{1}_{s=s'} & \text{otherwise} \end{cases}$$

where we define $|\boldsymbol{P}(s)| = \sum_{a \in Act} |\boldsymbol{P}(s)(a)|$, and $\mathbf{1}_c$ is an indicator value, equal to 1 if c is true, and to 0 otherwise. It is straightforward to map a labelled DTMC $M = (S, Act, \boldsymbol{P}, \iota)$ into a DTMC $M' = (S, \boldsymbol{P}', \iota)$, by defining $\boldsymbol{P}'(s)(s') = \sum_{a \in Act} \boldsymbol{P}(s)(a)(s')$. Note that this conversion is just a modelling shorthand in PRISM, and we do not do this if the model is intended to be an MDP.

We can *compose* MDPs (corresponding to PRISM modules) by forcing them to synchronise over certain actions. For MDPs $M_1 = (S_1, Act, \boldsymbol{P}_1, \iota_1)$ and $M_2 = (S_2, Act, \boldsymbol{P}_2, \iota_2)$ that have the same actions, we can define the composed MDP as $M_1 \| A \| M_2 = (S_1 \times S_2, Act, \boldsymbol{P}, (\iota_1, \iota_2))$, for $A \subseteq Act$, where $\boldsymbol{P}$ is defined as:

$$\boldsymbol{P}(s_1, s_2)(a) = \begin{cases} \boldsymbol{P}_1(s_1)(a) \otimes \boldsymbol{P}_2(s)(a) & \text{if } a \in A \\ \begin{array}{c} \boldsymbol{P}_1(s_1)(a) \otimes \{ \boldsymbol{\pi}_{s_2} \} \cup \\ \{ \boldsymbol{\pi}_{s_1} \} \otimes \boldsymbol{P}_2(s_2)(a) \end{array} & \text{otherwise} \end{cases}$$

where $\boldsymbol{\pi}_s$ is the distribution defined as $\boldsymbol{\pi}_s(s') = \mathbf{1}_{s=s'}$. We define the operator $\otimes$ over sets:

$$\begin{aligned} &\boldsymbol{P}_1(s_1)(a) \otimes \boldsymbol{P}_2(s)(a) = \\ &\qquad \{ \boldsymbol{\pi}_1 \otimes \boldsymbol{\pi}_2 \mid \boldsymbol{\pi}_1 \in \boldsymbol{P}_1(s_1)(a), \boldsymbol{\pi}_2 \in \boldsymbol{P}_2(s_2)(a) \} \end{aligned}$$

and over distributions:

$$(\boldsymbol{\pi}_1 \otimes \boldsymbol{\pi}_2)(s_1, s_2) = \boldsymbol{\pi}_1(s_1) \cdot \boldsymbol{\pi}_2(s_2)$$

In PRISM, it is possible for the guards of one module to look at the variables in another module — although in an update, a module can only modify its own variables. This makes the composition of modules slightly more complex than the above (see [1]), but the above is sufficiently general for this paper.

A. The LMAC Protocol for Wireless Sensor Networks

As a running example for the methodology presented in this paper, we take the Lightweight Medium Access Control (LMAC) protocol for wireless sensor networks [15]. This is a protocol designed for sensor networks where each node has only a limited amount of power, and therefore a limited communication range. When nodes are within range of one another, only one can transmit at a time, otherwise their transmissions will interfere (we call this a *collision*).

To avoid collisions, time is segmented into recurring *time frames*, which each consist of t fixed-length *time slots*. This is known as Time Division Multiple Access (TDMA). Each node has ownership of a time slot, in which it can transmit messages, but since we do not know the topology in advance, we need a method for negotiating which node gets which time slot. This is the purpose of the LMAC protocol, which uses a distributed algorithm to achieve this goal. We will only describe the part of the protocol concerned with negotiating time slots — routing messages across the network is straightforward once the time slots have been correctly allocated.

A special node called the *gateway node* initiates the LMAC protocol by selecting a time slot, and being the first to transmit any messages. The basic idea is then to have three phases of behaviour for the nodes in the network. When a node receives its first message, it enters the *scanning mode* where it listens over an entire time frame for any messages from its neighbours. Each node must transmit a short control message at the start of the time frame it owns (even if it has no data to send), which allows scanning nodes to determine which time slots are currently being used. Importantly, a node cannot have the same time slot as any of its neighbours *or* its neighbours' neighbours, since the latter would imply that one of its neighbours will detect a collision.

After the scanning mode, the node chooses one of the available time slots, and enters the *normal mode* of operation. There are two possibilities in this node, for each time slot. If it does not own the time slot, it listens for any messages. If it detects a collision (i.e. there is interference), it remembers this, so that it can notify its neighbours. At the time slot that the node owns, it sends a control message indicating the time slots it knows are in use, and whether it detected a collision in a time slot. Note that nodes cannot detect their own collisions, since they cannot transmit and receive at the same time.

If a node gets notified of a collision in the time slot it owns, it enters the *back-off mode* of operation. Here, it chooses a number of time frames to wait (in our model, this is between 0 and 3 frames), before entering the scanning mode once again.

We have built a model of the LMAC protocol in PRISM, parameterised by the topology of the network — we have a tool that takes a topology description as input, and generates the concrete PRISM model. The decisions of which time slot to choose (when more than one is available), and how long to

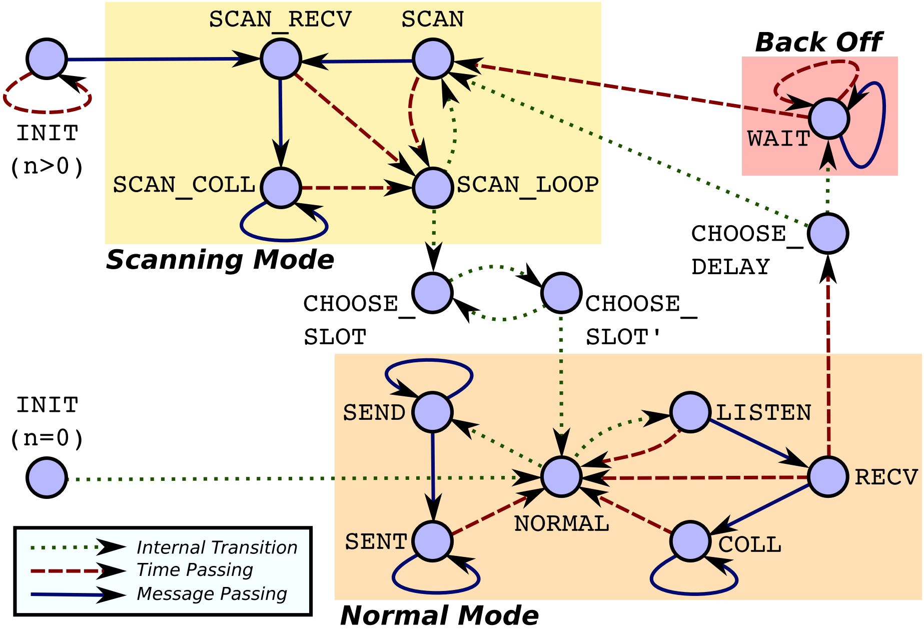

Fig. 1. State machine of a node in the LMAC protocol

back-off for are made probabilistically, and these probabilities are also parameters to the model. We model each node as a PRISM module, which has a number of variables recording, amongst other things, the state the node is in, which time slot it owns, the time slots of its neighbours, and whether it detected a collision. Figure 1 shows a state machine for a node, indicating how it moves between different modes of operation.

Notice that there are three types of transition in the model. *Internal transitions* (dotted arrows) are actions made by the node independently of the rest of the network — they correspond to internal decisions. *Message-passing transitions* (solid arrows) are actions corresponding to sending or receiving a message — the node (i.e. the PRISM module) synchronises with all and only its neighbouring nodes whenever it sends a message. Finally, *Time-passing transitions* (dashed arrows) correspond to the whole network moving to the next time slot, and all nodes must synchronise over these actions. The LMAC protocol assumes that there is a globally synchronised clock (i.e. framing is handled by a lower-level protocol), and we model this as a separate module (t is a parameter to the model — the number of time slots in a frame):

```
module Clock
    now : [0..t − 1] init 0 ;
    [tick]  true  →  (now′ = (now + 1) mod t)
endmodule
```

There is a link here with probabilistic timed automata [12], but we explicitly model a clock as a component in the model, rather than it being intrinsic to the modelling formalism.

The LMAC protocol was first formalised in [9], as a purely qualitative model in UPPAAL (based on timed automata), to verify whether the system eventually stabilises for a range of different topologies. They were able to consider all topologies up to five nodes, and their use of a clock in the UPPAAL model corresponds to our explicit modelling of time using a PRISM module. In [16] probabilities were added for the back-off times, but not for the time slot selection, resulting in an underlying semantics of a Markov Decision Process (MDP) — we have gone the extra step of making the model purely probabilistic.

More recently, a much higher-level model of the LMAC protocol has been developed, using combinatorial arguments to specify a DTMC for the evolution of the entire network [7]. Rather than modelling each node in detail, they just record how many nodes have a safe time slot, how many are colliding, and how many are backed-off for each number of frames. The model evolves in steps of one time *frame*, and so they do not model details such as messages being passed. Essentially, they aggregate the behaviour over all the time steps in the time frame into a single transition.

A limitation of this model is that since they rely on combinatorial arguments, they only consider clique topologies (where every node is within range of every other node). This gives a much more scalable model than more detailed approaches, but it strays somewhat from the intent of the protocol, where nodes have a limited communication range. Moreover, while they can consider in the order of 100 nodes, this also assumes that there are enough time slots (i.e. at least 100), whereas the LMAC specification suggests 32 time slots for practical purposes. If we put aside these concerns, however, we still have an important question to ask — is their model *correct*? Since many important details of the system have been abstracted away, it is difficult to be certain that no modelling artifacts were introduced.

This case study provides one of the main motivations of this paper — to automate the process of abstracting our detailed

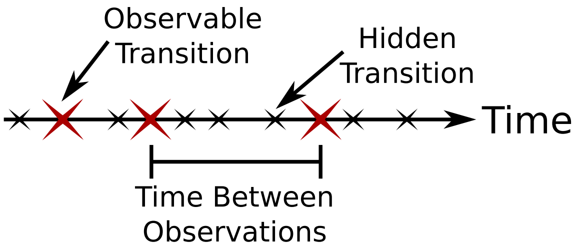

Fig. 2. Observable and hidden transitions in a DTMC

model so that it has the *same state space* as that of [7], enabling us to *compare* the two models, and detect modelling artifacts in the latter. We will now describe our approach.

III. Temporal Abstraction

In a DTMC, there is no notion of time. This means that there is no in-built way of modelling the duration of a transition. One common interpretation of a DTMC is to say that all transitions have the same duration, and this duration is deterministic. However, we may wish to build a more refined notion of time into the model — for example, that duration is a property only of certain *observable events* in the model, rather than a property of every transition.

This idea is illustrated in Figure 2. A DTMC does not tell us anything about *when* transitions take place; just the temporal *order* in which they occur. If some transitions are hidden from us, it does not make sense to talk about the time at which they occur — only that they take place at some point between two observable transitions. We can therefore talk about the time between observable transitions (which may be governed by some distribution), but not the duration of an individual transition. To state this concept more precisely, we can separate transitions into three types:

1) **Observable transitions**, which occur at a known point in time — we can think of an observable transition as fixing a global time for the entire model when it occurs. For the LMAC case study we present in this paper, it is useful to consider the time between two observable transitions to be deterministic (i.e. the length of a time slot), but we could choose other distributions, as we will discuss in the conclusions.
2) **Hidden transitions**, which happen between observable transitions, but at an uncertain point in time. These correspond to events where we are uncertain about the specific time at which they occur — except that they occur temporally before the next observable transition.
3) **Urgent transitions**, which occur instantaneously, as soon as they are enabled, with a higher priority than any other type of transition. These correspond to events that are not present in the real system, but are introduced for modelling purposes — usually, to make the model simpler or easier to describe.

An intuition for the distinction between observable and hidden/urgent transitions comes from synchronous digital electronics, where we only look at the voltages on output wires after a certain delay (the clock period) has passed. This is because we do not know how long the internal events take to happen (the propagation delays), as is the case with the models we are considering.

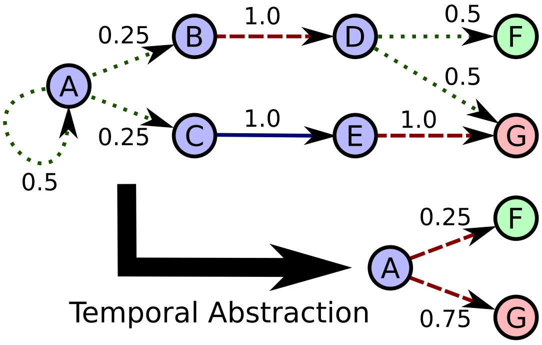

Fig. 3. An illustration of temporal abstraction

Note that there is some similarity here between the *vanishing* and *tangible* states of generalised stochastic Petri nets (GSPNs) [14], which roughly correspond to states with only urgent transitions or observable transitions respectively. The main difference is that there is no distinction between transitions that take no time (are urgent) and those that take some time, but we do not know their duration (are hidden).

When we divide the transitions in the model into the above types, it becomes clear that it only makes sense to *observe* the state of the system immediately after an observable transition takes place. Otherwise, if we make an observation at a different point in time, we cannot know how many hidden transitions have taken place. Between observable transitions, the state of the system is uncertain, and so it makes sense to abstract away from this by *collapsing* all the transitions between two observable transitions.

This idea is illustrated in Figure 3. Starting from the initial state, we compute the probability of reaching a state by performing a sequence of hidden or urgent transitions, followed by a *single observable transition*, and then as many urgent transitions as we are able to perform. We end up in a *stable state* corresponding to the next observable state of the system. By repeating this process, we can construct a *temporal abstraction* of the model.

Let us now formally describe how to identify these three types of transition, and how to compute the transition probabilities in the time-abstracted model. We will describe this in terms of labelled DTMCs, *using the action labels* to identify the transition types. Consider a labelled DTMC $(S, Act, \boldsymbol{P}, \iota)$. We can describe the actions as the union of three disjoint sets (where we require Obs to be non-empty):

$$Act = Obs \cup Hid \cup Urg$$

Let us now define what we mean by a *stable state* s', given that we start in a state s. The intuition is that there is a path from s to s' such that (1) precisely one transition is observable, (2) all the transitions before it are either hidden or urgent, (3) all the transitions after it are urgent, and (4) there are no urgent transitions out of s'. Defining this formally:

$$\begin{aligned} \mathit{Stab}(s, s') \text{ iff } & \exists s_0, s_1, \ldots, s_n.\ s_0 = s \wedge s_n = s' \wedge \\ & (1)\ \exists 0 \leq i < n.\ \mathit{Tran}(s_i, \mathit{Obs}, s_{i+1}) \wedge \\ & (2)\ \forall 0 \leq j < i.\ \mathit{Tran}(s_j, \mathit{Hid} \cup \mathit{Urg}, s_{j+1}) \wedge \\ & (3)\ \forall i < k < n.\ \mathit{Tran}(s_k, \mathit{Urg}, s_{k+1}) \wedge \\ & (4)\ \forall a \in \mathit{Urg}, s'' \in S.\ \boldsymbol{P}(s')(a)(s'') = 0 \end{aligned}$$

where $\mathit{Tran}(s_1, A, s_2)$ is defined as:

$$\mathit{Tran}(s_1, A, s_2) \text{ iff } \exists a \in A.\ \boldsymbol{P}(s_1)(a)(s_2) > 0$$

We can then construct a temporal abstraction $M^\sharp = (S^\sharp, \boldsymbol{P}^\sharp, \iota)$, where $S^\sharp = \{\, s \mid \mathit{Stab}^*(\iota, s) \,\}$, for the reflexive and transitive closure Stab^* of Stab (the set of all stable states that are reachable from the initial state). $\boldsymbol{P}^\sharp$ is defined as:

$$\boldsymbol{P}^\sharp(s)(s') = \begin{cases} \Pr_s^{M^{[s]}}(\Diamond a_{s'}) & \text{if } \mathit{Stab}(s, s') \\ 0 & \text{otherwise} \end{cases}$$

where $a_{s'}(x)$ evaluates to true iff $x = s'$, and the reachability probabilities are computed over the modified DTMC $M^{[s]} = (S, \mathit{Act}, \boldsymbol{P}^{[s]}, \iota)$, with $\boldsymbol{P}^{[s]}$ defined as:

$$\boldsymbol{P}^{[s]}(s_1)(a)(s_2) = \begin{cases} \frac{1}{|\mathit{Act}|}\mathbf{1}_{s_1 = s_2} & \text{if } \mathit{Stab}(s, s_1) \\ \boldsymbol{P}(s_1)(a)(s_2) & \text{otherwise} \end{cases}$$

This makes all the stable states that follow s absorbing, allowing $\boldsymbol{P}^\sharp$ to be computed in terms of reachability probabilities.

We still have to be a little careful in computing $\boldsymbol{P}^\sharp$, however, since it might be possible to never reaching a stable state after leaving a state s. This would mean that $\boldsymbol{P}^\sharp(s)$ defines a sub-probability distribution over $S^\sharp$, and therefore $M^\sharp$ is not a DTMC. We could fix by adding an additional state to $S^\sharp$, and re-directing the remaining probability mass there. Usually, however, this scenario just means that there is a mistake in the model, since it implies that the system enters a state where it can never be observed again.

Note that we throw away the labels in the temporal abstraction, and so it yields an (unlabelled) DTMC. If we want to keep the labels in *Obs* then this is straightforward to do, but since we do not need them in this paper, the above is simpler.

The temporal abstraction we produce ensures that all transitions in the new DTMC correspond to an observable event. This does not necessarily guarantee a reduction in the size of the state space, but it can do for models with a particular structure. For example, if all components in the model synchronise over observable transitions, and no other transitions can take place if an observable one is enabled (at the system level), then we can be sure to achieve a reduction in the size of the state space. This is the case in our model of the LMAC protocol, if we view the *tick* transitions as the observable ones. Specifically, we can classify the transitions in Figure 1 as follows:

1) The *observable transitions* are the time-passing transitions (with dashed arrows). These correspond to time globally passing (i.e. the clock ticking), and so these are the only points when we can be sure that the network is in a stable, observable state.
2) The *urgent transitions* are the internal transitions out of the states `SCAN_LOOP`, `CHOOSE_SLOT`, `CHOOSE_SLOT`′ and `CHOOSE_DELAY` (with dotted arrows). These should take place as soon as possible, since they are are only present to make the model description more compact.
3) The *hidden transitions* are the message passing transitions (with solid arrows), and the remaining internal transitions (with dotted arrows). These happen at some point within the time frame, but we do not know when, or in what order (for example, the precise order of messages may depend on many factors, and is not something we want to observe).

A. Controlling the Temporal Granularity by Sampling

Whilst the idea of a temporal abstraction is useful in restricting our view of the system to observable events, we may also want to take things one step further. Rather than viewing *every* observable event that takes place in the model, we might only want to *sample* these observations at a certain frequency. This makes a lot of sense in the LMAC protocol, when there are two levels of time granularity: *time slots*, and *time frames*. The detailed model describes how the system evolves over a time slot, but in the high-level model of [7] we look at how the entire network evolves over an entire time frame. To this end, we would like to abstract away from the detail of which time slots messages are sent in.

Luckily, it is very straightforward to change the temporal granularity of our abstracted model, given that we only look at its state every t events. Given a temporal abstraction $M = (S, \boldsymbol{P}, \iota)$, we can compute the t-step sampling as:

$$M_t = (S, \boldsymbol{P}^t, \iota)$$

For our LMAC model, t is simply the number of time slots in a time frame.

In practice, we do not want to compute this matrix power directly, but we can instead take advantage of $\boldsymbol{P}$ being a sparse matrix (as is usually the case when we build a compositional model in a language such as PRISM). If we only compute the transitions for the reachable states, then we can exploit the fact that the size of S is reduced. Note however that this size reduction *only happens if the sampling periodicity matches a periodicity in the behaviour of the model* — intuitively, allowing us to always skip over certain states in the model.

An important feature of both the temporal abstraction and the sampling is that we can perform it *on-the-fly* whilst we explore the state space of the detailed model. The size of the state space reduction can be seen in Table I, which we will discuss in more detail in Section V.

B. Behavioural Equivalences and Temporal Abstraction

The idea that we have presented in this section very much relates to ideas of behavioural equivalence, in the sense that our temporal abstraction is *trace equivalent* to the original model, if we collapse traces in the original model so that they only contain the observable states.

Let us formalise this statement more precisely. It is standard to define a probability space over the *cylinder sets* of a DTMC [3]. Namely, for a DTMC $M = (S, \boldsymbol{P}, \iota)$, and a finite path $\sigma = \iota, s_1, \dots, s_n$, we define:

$$\mathit{Cyl}(\sigma) = \{\, \sigma' \in \mathit{Paths}(M) \mid \sigma \text{ is a prefix of } \sigma' \,\}$$

We can define a probability measure over the σ-algebra of all cylinder sets of M, such that $\Pr^M(\mathit{Cyl}(\iota, s_1, \dots, s_n)) = \boldsymbol{P}(\iota, s_1) \cdots \boldsymbol{P}(s_{n-1}, s_n)$.

To allow us to compare paths in two different DTMCs, let us define a projection function over paths. Given a subset $S^\sharp$ of a state space S, we define $\rho_{S^\sharp}$ over finite paths in S:

$$\rho_{S^\sharp}(\epsilon) = \epsilon \qquad \rho_{S^\sharp}(s, \sigma) = \begin{cases} s, \rho_{S^\sharp}(\sigma) & \text{if } s \in S^\sharp \\ \rho_{S^\sharp}(\sigma) & \text{otherwise} \end{cases}$$

The correctness of our temporal abstraction is then expressed by the following theorem:

Theorem 4. *Let $M = (S, \mathit{Act}, \boldsymbol{P}, \iota)$ be a labelled DTMC, and $M^\sharp = (S^\sharp, \boldsymbol{P}^\sharp, \iota)$ the temporal abstraction of M with respect to $\mathit{Act} = \mathit{Obs} \cup \mathit{Hid} \cup \mathit{Urg}$. If M' is the (unlabelled) DTMC induced by M, then for all finite paths $\sigma^\sharp \in \mathit{Paths}(M^\sharp)$:*

$$\Pr^{M^\sharp}(\mathit{Cyl}(\sigma^\sharp)) = \Pr^{M'}(\cup\{\, \mathit{Cyl}(\sigma) \mid \rho_{S^\sharp}(\sigma) = \sigma^\sharp \,\})$$

Proof: We rely on the correct computation of the reachability probabilities $\Pr_s^{M^{[s]}}(\lozenge a_{s'})$. For $\sigma^\sharp = \iota, s_1^\sharp, \dots, s_n^\sharp$:

$$\begin{aligned} \Pr^{M^\sharp}(\mathit{Cyl}(\sigma^\sharp)) &= \boldsymbol{P}^\sharp(\iota, s_1^\sharp) \cdots \boldsymbol{P}^\sharp(s_{n-1}, s_n) \\ &= \Pr_\iota^{M^{[\iota]}}(\lozenge a_{s_1^\sharp}) \cdots \Pr_{s_{n-1}^\sharp}^{M^{[s_{n-1}^\sharp]}}(\lozenge a_{s_n^\sharp}) \end{aligned}$$

But for any path σ' such that $\rho_{S^\sharp}(\sigma') = \sigma^\sharp$, and for all pairs of states $s_i^\sharp$ and $s_{i+1}^\sharp$, there are no states s between these in σ' such that $\mathit{Stab}(s_i^\sharp, s)$. The above therefore corresponds to:

$$\begin{aligned} &\Pr^{M'}(\{\, \sigma \mid \exists \text{ prefix } \sigma' \text{ of } \sigma.\ \rho_{S^\sharp}(\sigma') = \sigma^\sharp \,\}) \\ &\qquad = \Pr^{M'}(\cup\{\, \mathit{Cyl}(\sigma) \mid \rho_{S^\sharp}(\sigma) = \sigma^\sharp \,\}) \end{aligned}$$

■

It is known that both strong and weak probabilistic bisimulation are stronger than trace equivalence [2], and in fact these are too strong for our purposes. In particular, PCTL properties are clearly not preserved by our abstraction, since they can 'view' the hidden states, and not just the observable ones. It is reasonable to suggest, however, that we could characterise our abstraction using a form of probabilistic *testing equivalence* [5] — in particular, if we restrict the tests so that they only concern observable states and transitions.

IV. Spatial Abstraction

We have now seen how to perform a temporal abstraction of a DTMC model such that only the transitions we want to *observe* are present in the abstract model. Furthermore, by *sampling* the model at different rates, we can adjust the *temporal granularity* of our abstraction. If we want to compare our model with a higher-level model of the same system, however, this temporal abstraction only takes us part of the way in itself — our model still contains a lot of detail about the *state* of each of the components that may be ignored in the high-level model.

In general, we want to *extract* certain properties of the states of our detailed model, and only record these in the states of the abstract model. For example, turning to our LMAC case study, we might want to record just the number of nodes that are colliding, rather than the specific time slot owned by each node in the network. In this section, we describe a *spatial* abstraction of models, which allows us to project onto such higher-level state spaces, with the trade-off that we may introduce additional non-determinism into the model.

Over the years, a number of techniques have been developed that allow us to do just this — for example, abstracting to Interval Markov Chains [8], [13], or MDPs [6]. We take the latter approach in this section, projecting a DTMC onto a smaller state space and thus constructing an MDP.

To perform a spatial abstraction on a DTMC $M = (S, \boldsymbol{P}, \iota)$, where $S \subseteq V_1 \times \cdots \times V_n$, we first define an *abstract state space* $S^\sharp = V_1^\sharp \times \cdots \times V_m^\sharp$. We do so by means of extraction functions $\eta_1, \dots, \eta_m$ of the form:

$$\eta_i : V_1 \times \cdots \times V_n \to V_i^\sharp$$

Typically, we can express each η_i in a compact form — for example, defining an abstract variable as the sum of the values of a number of the concrete variables. We define the extraction of a concrete state $s \in S$ as:

$$\eta(s) = (\eta_1(s), \dots, \eta_m(s))$$

The idea is that the abstract state provides a *summary* of the information in the concrete state. Often this corresponds to recording just the number of components in a given state, rather than the actual state of each individual component, hence we call this a *counting abstraction*.

Given a DTMC $M = (S, \boldsymbol{P}, \iota)$ and an extraction $(S^\sharp, \eta)$, we can define an MDP $M^\sharp = (S^\sharp, \boldsymbol{P}^\sharp, \eta(\iota))$ as:

$$\boldsymbol{P}^\sharp(s^\sharp) = \{\, \eta(\boldsymbol{P}(s)) \ \mid \ \eta(s) = s^\sharp \,\}$$

where we define an extraction of a distribution $\eta(\boldsymbol{\pi})$ as:

$$\eta(\boldsymbol{\pi})(s^\sharp) = \sum_{s \mid \eta(s) = s^\sharp} \boldsymbol{\pi}(s)$$

Note that this definition of an MDP differs from that given in Definition 2 in that there are no actions. We write $(S^\sharp, \boldsymbol{P}^\sharp, \eta(\iota))$, where $\boldsymbol{P}^\sharp : S^\sharp \to \mathcal{P}(S^\sharp \to [0,1])$, as a shorthand for $(S^\sharp, \{\, \tau \,\}, \boldsymbol{P}'^\sharp, \eta(\iota))$, where $\boldsymbol{P}'^\sharp : S^\sharp \to \{\, \tau \,\} \to \mathcal{P}(S^\sharp \to [0,1])$ — $\boldsymbol{P}'^\sharp(s^\sharp)(\tau) = \boldsymbol{P}^\sharp(s^\sharp)$. A further point is that there may be states in $s^\sharp \in S^\sharp$ that are unreachable from the initial state $\eta(\iota)$, and so we will typically only construct the reachable subset of $S^\sharp$.

As an example of a spatial abstraction, consider our LMAC model. In the high-level model of [7], they define the state space in terms of just five variables: the number of nodes with a safe time slot (*num_safe*), the number in the scanning mode (*num_scan*), and the number that have backed off for $1 \leq$

$k \leq 3$ time frames ($\mathit{num\_boff}_k$). We can define an extraction function for each of these, by expressing their value as an expression of the variables in the detailed model.

The variables $\mathit{num\_scan}$ and $\mathit{num\_boff}_k$ are straightforward to define for a model with N nodes, where we use the notation v_i to refer to variable v in the module for node i:

$$\mathit{num\_scan} = \sum_{i=0}^{N-1} \mathbf{1}[\ \mathit{state}_i = \texttt{SCAN}\ ]$$
$$\mathit{num\_boff}_k = \sum_{i=0}^{N-1} \mathbf{1}[\ \mathit{state}_i = \texttt{WAIT} \wedge \mathit{counter}_i = k\ ]$$

where we write the indicator value $\mathbf{1}_c$ as $\mathbf{1}[c]$ for readability.

The variable $\mathit{num\_safe}$ is a little trickier to define, since it depends on the topology of the network. A node has a safe time slot if it is in the normal mode, and none of its neighbours or its neighbours neighbours are also in normal mode with ownership of the same time slot. Given an edge relation E describing the connectivity of the nodes:

$$\mathit{num\_safe} = \sum_{i=0}^{N-1} \mathbf{1}[\ \mathit{state}_i = \texttt{NORMAL} \wedge \neg\exists j \neq i.$$
$$((i,j) \in E \vee \exists k.\ \{\,(i,k),(k,j)\,\} \subseteq E) \wedge$$
$$\mathit{slot}_i = \mathit{slot}_j \wedge \mathit{state}_j = \texttt{NORMAL}\ ]$$

Notice that we can infer the number of nodes that have a colliding time slot from these variables:

$$\mathit{num\_coll} = N - \mathit{num\_safe} - \mathit{num\_scan} - \sum_{k=1}^{3} \mathit{num\_boff}_k$$

An important feature of the spatial abstraction is that, like the temporal abstraction, we can construct it *on-the-fly* as we explore the state space of the detailed model. Intuitively, we can construct the MDP as we go along by adding one probability distribution to a state at a time — introducing a new non-deterministic choice in the MDP whenever we add a distribution to a state that does not coincide with any existing possibility. By doing this on-the-fly we can avoid storing the transitions in the original and temporally-abstracted models, which leads to a large reduction in memory consumption.

V. LMAC: A Case Study

To apply our abstraction methodology, we have developed a tool, written in OCaml, which takes a PRISM model as input, along with a definition of which transitions are observable, hidden and urgent, and an abstraction file that specifies the variables in the high-level model, as expressions of the original variables. The output is an MDP (the temporally and spatially abstracted model) in a format readable by PRISM, so that we can use its model-checking engine to verify properties of it.

Table I shows the time and memory usage of our tool for various instantiations of the LMAC model. This was run on a MacBook with a 2.13 GHz Intel Core 2 Duo processor, and 4 GB of RAM. As can be seen, we can handle models with up to 10^9 states on this architecture. Note that on the same architecture, PRISM was only able to build the state space for models with 4 nodes (i.e. up to around 100,000 states), even after reordering the variables in the model. We suspect that because the models are complicated (even the clique topology for 4 nodes has 53 variables and 349 commands), the symbolic MTBDD representation used by PRISM is inefficient and incurs a lot of overhead. We discovered a remarkable improvement by implementing an explicit bit-vector representation in our tool.

The reason why we can keep the memory consumption so low is that we compute the abstraction *on-the-fly*. This means that we do not store every state we explore — only the ones after the temporal abstraction. Further to this, we can also construct the transitions in the spatial abstraction on-the-fly, so we do not need to store the transitions in the temporal abstraction. This means that we only need a single hash table to record the states that we have visited while constructing the temporal abstraction, which allows us to abstract very large models. It should be noted that the largest models of LMAC that the authors of [9] were able to analyse contained just five nodes, and these were qualitative models (in UPPAAL). We are able to analyse *quantitative* models of networks with six nodes, and even seven nodes under certain topologies, hence our abstraction enables us to out-perform existing techniques.

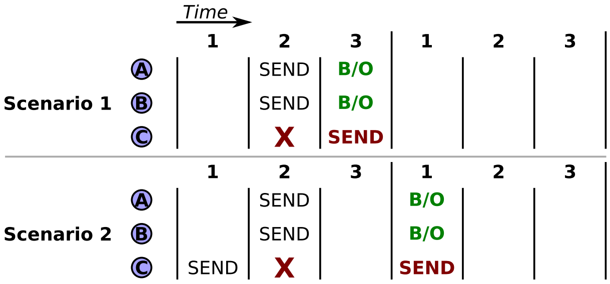

Fig. 4. Non-determinism in the abstracted LMAC model

A. Investigation of Modelling Artifacts

If we look at the MDPs our tool produces from the detailed LMAC models, we can *directly compare* the output to the high-level model of [7]. Complete agreement would mean that the MDP we produce has only one choice in every state (i.e. there is no non-determinism), and that the two DTMCs are isomorphic. If this were the case, we could argue that the high-level model is free of modelling artifacts, and accurately describes the behaviour of the protocol. This is not quite the case, however, as we shall now discuss.

We were able to find schedulers for the MDPs we produce from clique topologies that agree with the models in [7] (note that we cannot compare the tree topologies, since these are not considered in [7]). This validates the combinatorial arguments that they use in their model. However, we found that our abstractions contain non-determinism, which means that the models do not entirely agree. This arises because the abstract model does not have enough information to know how long it takes for a node to be informed that it is in a collision.

Topology	Nodes	Slots	Variables/ Commands	States Explored	States After Temporal Abstraction	States After Spatial Abstraction	Time (s)	Memory (MB)
Clique	4	4	53/349	37,055	829	39	1	15
	5	5	76/611	1,794,265	22,657	86	62	19
	6	6	103/979	156,361,383	980,766	174	7,759	351
Tree	7	6	120/521	79,048,156	609,423	101	4,300	322
	7	7	134/569	230,930,452	1,352,224	101	13,086	686
	7	8	148/617	579,819,743	2,716,577	101	31,992	1500

TABLE I
RESULTS FOR ABSTRACTION OF DIFFERENT LMAC TOPOLOGIES

Figure 4 shows two scenarios for a clique topology with three nodes and three time slots. In the first scenario, nodes A and B own time slot 2, and node C owns time slot 3. A and B collide in time slot 2, and this is detected by node C, who informs A and B in time slot 3, allowing the nodes to immediately enter the back-off mode. Hence, at the end of the first time frame, we observe A and B in the *back-off mode*.

In the second scenario, node C owns time slot 1. This means that when C sends its message, it has not observed the collision, since it has not yet happened. This means that A and B must wait until the *next time frame* to be notified of their collision, and at the end of the first time frame, we observe A and B in the *normal mode*, but colliding.

This leads to non-determinism in the abstract model because we throw away information about which node owns which time slot — hence, both scenarios can be possible in the same abstract state. In [7], they assume that it always takes one time frame for a node to be notified of a collision (i.e. the second scenario), hence they over-approximate this aspect of the model. Since they did not consider such details when working at a high level of abstraction, they inadvertently introduced a *modelling artifact* into their model.

Rather than just examining the models that our abstraction produces, it is more interesting to analyse some properties of them. One property of particular interest for LMAC is the number of time frames that it takes for the network to stabilise — in other words, for all N nodes to own a safe time slot. We can express the cumulative probability up to time frame T as the following reachability property, where $\iota^\sharp$ is the start state of the abstract model:

$$\Pr_{\iota^\sharp}(\Diamond^{\leq T} \mathit{num\_safe} = N)$$

Figures 5 and 6 show the results of using PRISM on the MDP that our tool produces, to model check this property (for T ranging from 0 to 20) for the clique topologies with 4 and 5 nodes respectively. The upper and lower bounds are very close when we have 4 nodes, yet the lower bound becomes significantly worse when we move to 5 nodes — in fact, it gets worse the more nodes we add!

The reason for this only becomes clear when we take a careful look at the behaviour of the model. When there are 4 nodes in the network, it is only ever possible to have a collision in one time slot — the gateway node never collides, and so a collision is due to either all three remaining nodes colliding in the same time slot, or just two of them colliding. When we move to 5 nodes, it becomes possible for two collisions in two different time slots to occur, and this makes a big difference!

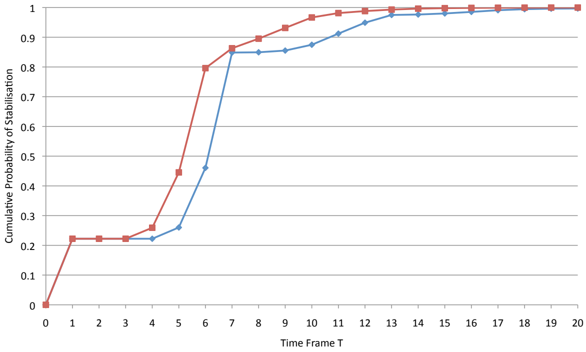

Fig. 5. Upper and lower bounds for the time to stabilise for a 4 node clique

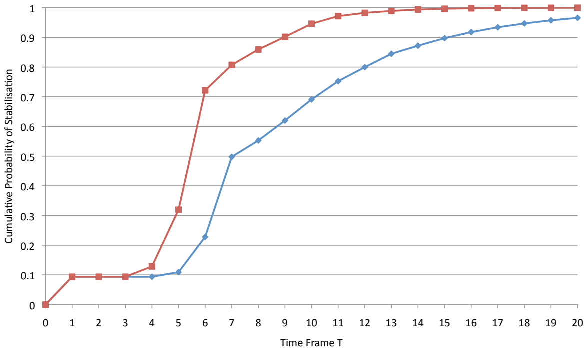

Fig. 6. Upper and lower bounds for the time to stabilise for a 5 node clique

If two such collisions occur, the two sets of nodes responsible can be notified of this in *different time slots*, and therefore back-off at different times. This means that it is possible to have two nodes in the scanning mode at the same time, but offset by one time slot. Consider a scenario with just two time slots available. The first node might enter the normal mode during the last time slot in a frame, choosing one of the slots. At the end of the frame, we do not observe any collisions. However, in the next time slot (which is in the next frame) the second node chooses its slot — still thinking that there are two available, because the first node has yet to send anything!

In such a scenario, we have a collision probability of 50%, but in the abstract model we cannot distinguish between this and the more common scenario when the node has correct information, and a 0% chance of colliding. This explains why the lower bound gets worse — we always have to consider the worst-case scenario where the collision probability is higher. This is another example of a modelling artifact, which suggests that the high-level model of [7] should be enriched with some additional details to distinguish these situations. Note that we see similar results regarding the lower bound for the tree topologies, but with a faster time to stabilise in the best case.

It is important to understand that the non-determinism introduced by the spatial abstraction, leading to the uncertainty in the above examples, is *not* a modelling artifact. The abstraction is correct with respect to the behaviour of the detailed model, and so the uncertainty simply indicates that the particular choice of spatial abstraction was not a good one. In other words, the high-level model has not taken into account some situations that occur in the detailed model. Using this knowledge, we can improve the high-level model, and therefore control the modelling artifacts within it.

VI. Conclusions

In this paper, we have described a novel methodology that allows us to take a detailed model of a system, and abstract it both *temporally* and *spatially* to build a higher-level mode. By comparing this with high-level models that are constructed by other means (e.g. in [7]), we can identify *modelling artifacts* that were introduced due to incorrect assumptions. We have illustrated our approach with a case study of the LMAC protocol for wireless sensor networks, showing that our methodology is both successful at identifying modelling artifacts, and in allowing us to analyse much larger models than are possible without abstraction.

There is a great deal of scope for future work, based on the ideas we have presented. From a practical side, we could look to scale our tool further by distributed computation, and investigate the possibility of computing some of the abstraction steps compositionally. There are also important similarities with the idea of on-the-fly model checking [4], since we construct our abstraction on-the-fly. More importantly, however, our methodology automates the building of high-level models from detailed ones, and so to use this to improve the high-level models we build, we need techniques for generalising the results we observe for small systems to much larger ones.

The methodology we described in this paper was limited to discrete-time models, but there are some straightforward extensions we can apply to move to continuous time. If the time between two observable transitions is distributed according to a random variable $X \sim Exp(\lambda)$, for some rate λ, then it is straightforward to construct a *continuous time MDP* (a CTMDP) as an abstraction of the model.

In doing this, the only place to be careful is when we adjust the temporal granularity — if we sample every t transitions, then the duration of a transition in the abstract model is Erlang-distributed: $X_t \sim \Gamma(t, \lambda)$. To construct a CTMDP, this means that we have to replace each state in the abstract model with a sequence of t states. But since the abstract models are typically very small, as is t, this should not present much difficulty in terms of the model size. Note that this yields a uniformised CTMDP (with rate parameter λ), for which efficient model-checking algorithms exist.

In summary, we have demonstrated through a methodical application of abstraction steps that it is possible to control modelling artifacts when we move to higher levels of abstraction. We can never eliminate them entirely, but if we better understand how modelling artifacts arise, we can improve our models and be more confident in the predictions they give.

Acknowledgements

This work was supported by MT-LAB, a VKR Centre of Excellence, and by the Danish Research Council (FTP grant 09-073796).

References

[1] The PRISM language — semantics. www.prismmodelchecker.org/doc/semantics.pdf.

[2] C. Baier and H. Hermanns. Weak bisimulation for fully probabilistic processes. In *9th International Conference on Computer Aided Verification*, volume 1254 of *LNCS*, pages 119–130. Springer, 1997.

[3] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.

[4] I. Beer, S. Ben-David, and A. Landver. On-the-fly model checking of RCTL formulas. In *10th International Conference on Computer Aided Verification*, volume 1427 of *LNCS*, pages 184–194. Springer, 1998.

[5] I. Christoff. Testing equivalences and fully abstract models for probabilistic processes. In *CONCUR '90 Theories of Concurrency: Unification and Extension*, volume 458 of *LNCS*, pages 126–138. Springer, 1990.

[6] P. D'Argenio, B. Jeannet, H. Jensen, and K. Larsen. Reduction and refinement strategies for probabilistic analysis. In *Process Algebra and Probabilistic Methods: Performance Modeling and Verification*, volume 2399 of *LNCS*, pages 335–372. Springer, 2002.

[7] L.J.R. Esparza, K. Zeng, and B.F. Nielsen. A probabilistic model of the LMAC protocol for concurrent wireless sensor networks. In *The 11th International Conference on Application of Concurrency to System Design (ACSD)*, 2011.

[8] H. Fecher, M. Leucker, and V. Wolf. Don't know in probabilistic systems. In *SPIN'06*, volume 3925 of *LNCS*, pages 71–88, 2006.

[9] A. Fehnker, L. van Hoesel, and A. Mader. Modelling and verification of the LMAC protocol for wireless sensor networks. In *6th International Conference on Integrated Formal Methods (IFM 2007)*, volume 4591 of *LNCS*, pages 253–272. Springer, 2007.

[10] H. Hansson and B. Jonsson. A framework for reasoning about time and reliability. *Proceedings of the Real Time Systems Symposium*, pages 102–111, Dec 1989.

[11] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: A tool for automatic verification of probabilistic systems. In *12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *LNCS*, pages 441–444. Springer, 2006.

[12] H.E. Jensen. Model checking probabilistic real time systems. In *7th Nordic Workshop on Programming Theory*, pages 247–261, 1996.

[13] B. Jonsson and K.G. Larsen. Specification and refinement of probabilistic processes. In *LICS '91: Proceedings of Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 266–277, 1991.

[14] M.A. Marsan, G. Conte, and G. Balbo. A class of generalized stochastic Petri nets for the performance evaluation of multiprocessor systems. *ACM Transactions on Computer Systems*, 2(2):93–122, 1984.

[15] L.F.W. van Hoesel and P.J.M. Havinga. A lightweight medium access protocol (LMAC) for wireless sensor networks: Reducing preamble transmissions and transceiver state switches. In *1st International Workshop on Networked Sensing Systems (INSS 2004)*, pages 205–208. Society of Instrument and Control Engineers (SICE), 2004.

[16] M.S. Vighio and A.P. Ravn. Analysis of collisions in wireless sensor networks. In *21st Nordic Workshop on Programming Theory*, 2009.