An Abstraction Relation for Energy Consumption Properties

Master student: Laurie Lugrin*
Supervisors: Florence Maraninchi* and Laurent Mounier*
* Verimag, 2, avenue de Vignate, 38610 GIÈRES – France
Email: ⟨firstname⟩.⟨lastname⟩ @imag.fr

***Abstract*—This paper deals with the validation of energy consumption properties for embedded systems. We are particularly interested in formal validation based on models for the validation of worst-case energy consumption. We restrict ourself to non-functional properties, that is, the system behaviour does not depend on the amount of energy consumed or left. In most cases models are too big to run verification, so we consider model abstractions.**

We use power state automata to model systems and an abstraction relation which is suitable to the worst-case consumption. We propose an automatic procedure to check whether a model is an abstraction of a given detailed model.

I. Introduction

Energy saving is becoming a major social concern this last few years and affects computer systems design. It is particularly important in embedded systems because they have a limited stock of energy. Energy saving increases the battery life, or the system lifespan in case the battery cannot be recharged.

The problem we are dealing with is that there is no well-established design methods to guarantee properties on energy consumption, so we turn towards ad-hoc methods. In this paper we choose to consider a formal approach in order to verify properties on energy, like worst-case lifespan of a system.

We use *power state automata* to model the system, a widely used formalism to express resource consumption. It consists of standard input/output automata (transductors) with state labels expressing the instantaneous energy consumption. We only consider systems whose behaviour does not depend on the amount of energy left or consumed. In this case, properties on energy are said to be non-functional. Our models fit our expressiveness need.

Several validation methods can be used to check properties on energy. In most cases we need to abstract the system first to allow verification within a reasonably short time.

An abstraction relation must suit the type of property we want to prove. Device providers are interested in guaranteeing a worst-case battery life, thus many relevant properties deal with worst-case energy consumption. Consequently an abstract model must over-approximate the consumption of the detailed one.

Moreover, as abstract models are usually built by hand, we need an automatic procedure to decide whether a model is an abstraction of another one.

At last, systems are often designed component-wise because they are too big to be designed as a single flat automaton. The abstraction relation must be modular to cope with this fact. It guarantees that the composition of abstract models of components is an abstraction of the complete detailed model.

We use an abstract relation which observes these criteria and which was first proposed by Ludovic Samper in [1]. He proved this relation to be a congruence w.r.t. the synchronous composition. To make this abstraction relation useful, the only thing left is to find an automatic decision procedure[1] for the abstraction relation.

A solution could consist in adapting general methods like abstract interpretation [2]. It would probably lead to a non exact procedure. We favour an ad-hoc procedure, based on graphs, which is exact and efficient.

The paper is structured as follows. First we present related works in Section II. Then we give formal definitions in Section III and propose a decision procedure for the abstraction relation in Section IV. Section V deals with some implementation details and Section VI concludes.

II. Related works

Our work pursues an earlier study of Ludovic Samper [1]. His topic was the computation of the worst-case lifespan of wireless sensor networks. He modelled components with power state automata and defined an abstraction relation which takes into account the environment, described by a language called *context*, in which components are used. He proved this relation to be a congruence w.r.t. the synchronous composition, which means that abstract models of components can be composed and form an abstract model of the global detailed model. A point is missing to make this work usable in practice: abstract models should be validated, either by construction or by an independent decision procedure. We complete his work with such automatic decision procedure.

The authors of [3] also use discrete models and abstractions to analyse costs. They focus on long-run cost. This work is hard to adapt to energy consumption of embedded systems because their battery can die long before the long-run consumption becomes a possible abstraction of the real consumption. Moreover, [3] only give a sufficient condition

[1]Notice that there are two validations in the verification process: first, abstract models are validated; and second the full model is validated against a property. We only deal with the abstract model validation.

for this approximation to be correct, whereas we have an exact answer.

Another energy representation of components could be *Linearly Priced Timed Automata* (LPTA) [4]. Originally these hybrid models were designed for scheduling purposes. There are minimum-cost algorithms for these models, but they are expensive in time and cannot be easily adapted to maximum-cost problems, which is our main interest to find a worst-case consumption.

III. Formal definitions

In this section, we define models we use, composition operators on them and the abstraction relation.

Two kinds of models are involved in the abstraction relation: component model and context model. A standard input/output automaton and a recogniser could be used to represent a component and a context, respectively. However, for simplicity sake, we use an automaton which is a mix between these two models. This makes the composition easier.

We call *input* a boolean formula over input signals. A *trace* is a finite sequence of inputs.

Definition ((deterministic) Mealy automaton)**.** *A Mealy automaton is tuple $\langle S, s_0, I, O, T, S_f\rangle$ where*

- *S is a finite set of states*
- *$s_0 \in S$ is the initial state*
- *I is a set of input signals*
- *O is a set of output signals*
- *$T \subseteq Q \times \mathbb{B}(I) \times \mathbb{B}(O) \times Q$ is a set of transitions. A transition (s, i, o, s') is written $s \xrightarrow{i/o} s'$*
- *$S_f \subseteq S$ is a set of accept states.*

such that for all state $s \in S$ and for all inputs $i \in \mathbb{B}(I)$ there is at most one transition $s \xrightarrow{i/o} s'$ for some $o \in \mathbb{B}(O)$ and $s' \in S$.

When representing a component, a Mealy automaton must be satisfy $S = S_f$ because all input traces are *a priori* possible. It must also be complete, i.e. for all states s and all inputs i there is a transition from s whose guard is i. When modelling a context, its set of output signals is empty.

Accepted trace and *language* are defined as for standard input/output automata.

Definition (Cost automaton)**.** *A cost automaton is a Mealy automaton $\langle S, s_0, I, O, T, S_f\rangle$ such that $S = S_f$, together with a cost function $C : S \to \mathbb{R}$ which labels each state with a cost.*

Cost automata are used to model components. The cost represents the instantaneous energy consumption.

Notice that energy does not appear on guards of cost automata. It is consistent with our hypothesis of non-functional properties: the system does not change its behaviour according to the energy.

Definition (synchronous product for Mealy automata)**.** *Let $A = (S^A, s_0^A, I^A, O^A, T^A, S_f^A)$ and $B = (S^B, s_0^B, I^B, O^B, T^B, S_f^B)$ be two Mealy automata.*

The synchronous product of A and B, noted $A \times B$ is the Mealy automaton

$$(S^A \times S^B, (s_0^A, s_0^B), I^A \cup I^B, O^A \cap O^B, T, S_f^A \times S_f^B)$$

where T is defined as follows:

$$(q_1^A, q_1^B) \xrightarrow{(i^A \wedge i^B)/(o^A \wedge o^B)} (q_2^A, q_2^B) \in T \Leftrightarrow$$
$$(q_1^A \xrightarrow{i^A/o^A} q_2^A) \in T^A \ \wedge \ q_1^B \xrightarrow{i^B/o^B} q_2^B \in T^B$$

We define some functions to observe automata executions.

Definition ($States$, Out, $Cost$)**.** *Let A be a Mealy automaton and $\langle A, C\rangle$ a cost automaton. Let t be a finite trace defined on the automaton input signals.*

$States(A, t)$ is the sequence of states passed through during the execution of A on t. $Out(A, t)$ is the sequence of outputs emitted during the execution of A on t.

Let $States(A, t) = [s_0, s_1, \ldots, s_n]$. Then $Cost(\langle A, C\rangle, t) = \sum_{j=0}^{n} C(s_j)$.

The synchronous composition for cost automata is the standard synchronous composition of the Mealy automata and an operation of the cost functions. This operation is a parameter and can be chosen depending on the use case.

Definition (synchronous product for cost automata)**.** *Let $M^A = \langle A, C^A\rangle$ and $M^B = \langle B, C^B\rangle$ be two cost automata, and $op : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ be a commutative and associative operator on real numbers.*

The synchronous product of M^A and M^B parameterised by op, written $M^A \times_{op} M^B$, is the cost automaton $M^D = \langle D, C^D\rangle$ where $D = (S, s_0, I, O, T, S_f)$ is defined as follows :

- *$D = A \times B$*
- *$\forall (a, b) \in S \quad C^D(a, b) = C^A(a)$ op $C^B(b)$*

In the sequel cost automata will be viewed as weighted directed graphs.

Definition (weighted directed graph)**.** *A weighted directed graph is a tuple $\langle V, E, w\rangle$ where*

- *V is a set of vertices*
- *$E \subset V \times V$ is a set of edges*
- *$w : E \to \mathbb{R}$ is a weight function.*

Definition (weighted directed graph induced by a cost automaton)**.** *Let $M = \langle (Q, Q_0, I, O, T, Q_f), C\rangle$ be a cost automaton. M induces a weighted directed graph, written $Graph(M) = (V, E)$ and defined as follows:*

- *$V = S \cup \{v_{init}\} \cup \{v_{final}\}$ where $\{v_{init}, v_{final}\} \cap S = \emptyset$*
- *$(v_1, v_2) \in E$ iff $\exists i \in \mathbb{B}(I) \cdot \exists o \in \mathbb{B}(O) \cdot v_1 \xrightarrow{i/o} v_2 \in T$*
- *$(v_{init}, s_0) \in E$*
- *$(v, v_{final}) \in E$ for all $v \in Q_f$*

The weight function w is defined as follows:

$$\forall (v_1, v_2) \in E, v_2 \neq v_{final} \quad w((v_1, v_2)) = C(v_2)$$

and

$$\forall (v_1, v_{final}) \in E \quad w((v_1, v_{final})) = 0$$

Notice that costs are on states in automata whereas they are on the incoming edges in graphs. Notice also that graphs have two particular vertices, the initial and the final ones. It will be useful to run classical graph algorithms.

We now give the definition of the abstraction relation, first introduced by Ludovic SAMPER in [1]. As usual abstraction relations, it sets that both models have the same outputs if they receive the same inputs, and that the abstract model over-approximates the detailed one, which means in our energy context that the abstract model consumes more than the detailed one.

An originality in Ludovic SAMPER's work was to parameterise the abstraction relation with a set of possible traces, for which the abstraction relation must hold. The other traces are considered as unrealistic and nothing is required for them. This avoids a relevant abstract model to be refused only because the abstraction does not hold for some unrealistic traces. The set of realistic traces is called *context*. We choose to describe it using a regular language. It is modelled by a Mealy automaton with an empty set of output signals.

Definition (Context-Based Abstraction Relation)**.** *Let $M^A = \langle A, C^A \rangle$ and $M^B = \langle B, C^B \rangle$ be two cost automata, and K a Mealy automaton. We say that M^B is an abstraction model of M^A under the context K, written $A \preceq_K B$, iff:*

$$\forall t \in \mathcal{L}(K), \begin{cases} Out(A,t) = Out(B,t) \\ \textit{and} \\ Cost(A,t,C^A) \leq Cost(B,t,C^B) \end{cases}$$

IV. DECISION PROCEDURE

We propose a procedure that decides, given two component models and a context model, whether one model is an abstraction of the other one under the context. We explain the idea of the procedure, before giving elements of the proof. Then we study the complexity of the procedure and finally explain how this procedure can be extended to produce a counter-example.

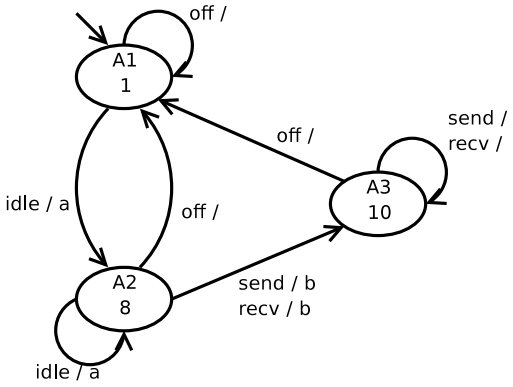
A running example is given in Fig 1.

A. Informal description

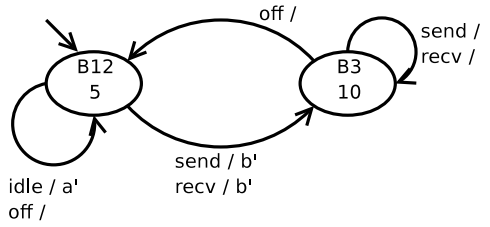
We want to compare two component models described by power state automata. Let's call the detailed model A and the supposed abstract model B. The context is noted K.

The question is whether A is an abstraction of B under the context K, written $A \preceq_K B$. We have to check the two following properties. The output property states that both component models should have the same outputs if they receive the same input trace and this trace is an accepted trace of K. The energy property states that the abstract model B should have a greater consumption than the detailed model A for all accepted traces of K.

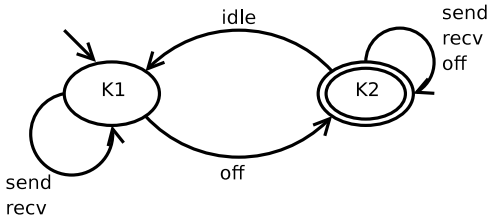
First, we want to compare A and B with respect to their outputs and energy consumption, when running with the same inputs. So the first step is to compose (by synchronous product) them so as to get an automaton which represents both automata in parallel.



(a) detailed model A



(b) abstract model, B



(c) context model, K

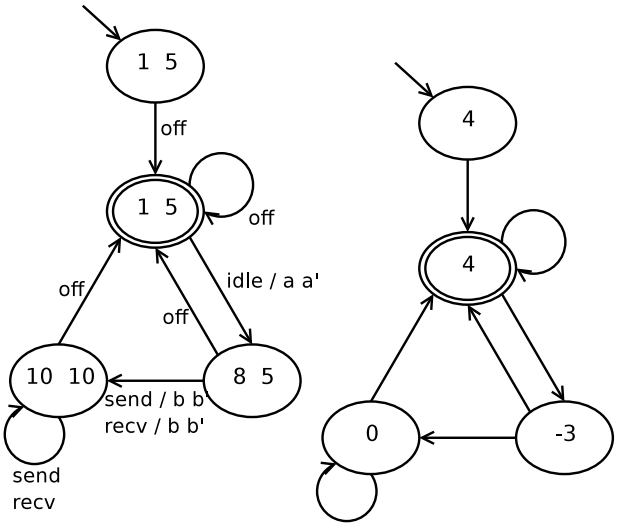
Figure 1. Data example

Second, we want to test outputs and energy consumption only for traces given in the context. To filter relevant paths, we compose the context with the previous product. Let's call the result P. The product for our running example is given in Fig 2(a).

Now, we have a unique condensed structure P with all the information we need. We have to test the two following properties: for each accepted trace of P, all transitions along the path which validates that trace are labelled with the same outputs for A and B; for each accepted trace of P, the sum of all costs collected along the path by B is greater or equal to the sum for A.

Clearly, we cannot consider each path individually because the number of paths may be infinite. We have to be crafty and find an equivalent global property on the automaton, for both properties.

For the property on outputs, we have to consider all transitions which can be taken while validating a trace, and only



(a) product automaton P (b) graph G induced by the product

Figure 2. Structures built to validate $A \preceq_K B$ (given in Fig. 1)

these ones. They are exactly the transitions which can be reached from an initial state and lead to an accept state. So, we can clean the automaton P, so as to remove all useless states and transitions, and then check each transition, one after the other. In the running example, the product of Fig 1 is already clean.

For the property on energy consumption, we consider only the difference of instantaneous consumptions between B and A. Now we want to check that for all accepted traces of P, the sum of the differences collected along the path is positive or null, that is B over-approximates the energy consumption of A. In other traces, we want to check that no path between an initial state and an accept state is strictly negative. We can simply run a lightest-path algorithm on the automaton, viewed as a weighted graph, and check that the lightest path has a positive weight. The graph for our running example is given by Fig 2(b).

B. Proof sketch

The proof is organised in successive formula equivalent to the definition of $A \preceq_K B$. The last formula is an effective decision procedure because it is computable.

The data are two cost automata $M^A = \langle A, C^A \rangle$ and $M^B = \langle B, C^B \rangle$, and a Mealy automaton K called context which recognises a non-empty language. An example of data is given in Fig 1.

The definition of $A \preceq_K B$ can be written as follows:

$\forall t \in K$
Let
$(a_j)_{j=0}^{j=n} = States(A,t)$
$(b_j)_{j=0}^{j=n} = States(B,t)$

$(o_j^A)_{j=0}^{j=n-1} = Out(A,t)$
$(o_j^B)_{j=0}^{j=n-1} = Out(B,t)$
In
$\forall j \in [\![0, n-1]\!] : o_j^A = o_j^B$
and
$\sum_{j=0}^{n} C^A(a_j) \leq \sum_{j=0}^{n} C^B(b_j)$

1) Product: Suppose that the detailed model A and the presumed abstract one B, as well as the context K, are complete. This means that for any state s and any input i there is a transition $s \xrightarrow{i/o} s'$ for some o and s'. If this is not the case, it is easy to make them reactive by adding sink states.

Intuitively, this implies that during the parallel execution, no automaton blocks the execution of another one. More formally, the following properties hold for the product $P = A \times B \times K$. First, as A and B have only accept states, the product P recognises the same language as K. Second, for all executions the output produced by the product P is the disjoint union of the outputs produced by A and B for the same trace. The product P for the running example is given in Fig 2(a).

We want to compare outputs of A and B on transitions of P but the product operator gathers outputs in a set. We can get round this problem by adding primes to B output signals before building the product, so as to keep all output signals side by side on the transitions. We introduce an equivalence function *equiv* which maps A outputs on B outputs to compare them.

The definition of $A \preceq_K B$ can be written equivalently this way:
Let
$P = A \times B \times K$
In
$\forall t \in P$
Let
$(a_j b_j k_j)_{j=0}^{j=n} = States(P,t)$

$(o_j^A o_j^B)_{j=0}^{j=n-1} = Out(P,t)$
In
$\forall j \in [\![0, n-1]\!] : \text{equiv}(o_j^A, o_j^B)$
and
$\sum_{j=0}^{n} C^A(a_j) \leq \sum_{j=0}^{n} C^B(b_j)$

2) Cleaning: The second step is the cleaning of the product: we remove states (and transitions) of the product P that cannot be reached, and those which cannot lead to a final state. Let's call $\tilde{P}$ the result.

The execution of an accepted trace on P only passes through states and transitions which are kept in $\tilde{P}$. Traces which are rejected by P are also rejected by $\tilde{P}$. So the cleaning operation does not alter executions on the product. In our running example, $\tilde{P} = P$. It is the case whenever A, B and K are clean.

Then we can use $\tilde{P}$ instead of P:

Let
  $\tilde{P} = \text{cleaning}(A \times B \times K)$
In
  $\forall t \in \tilde{P}$
    Let
      $(a_j b_j k_j)_{j=0}^{j=n} = States(\tilde{P}, t)$
      $(o_j^A o_j^B)_{j=0}^{j=n-1} = Out(\tilde{P}, t)$
    In
      $\forall j \in [\![0, n-1]\!] : \text{equiv}(o_j^A, o_j^B)$
      and
      $\sum_{j=0}^{n} C^A(a_j) \leq \sum_{j=0}^{n} C^B(b_j)$

3) Outputs: Let's focus on the output part of the property:

Let
  $\tilde{P} = \text{cleaning}(A \times B \times K)$
In
  $\forall t \in \tilde{P}$
    Let
      $(o_j^A o_j^B)_{j=0}^{j=n-1} = Out(\tilde{P}, t)$
    In
      $\forall j \in [\![0, n-1]\!] : \text{equiv}(o_j^A, o_j^B)$

We do not need to consider every accepted trace, we can simply consider every transition of the model P. Indeed, each transition of $\tilde{P}$ is on a path between the initial state and an accept state, so for each transition, there is at least one accepted trace passing through it, and then each transition must be such that its output signals are the same for A and B.

For the output part, we have to check:

Let
  $\tilde{P} = \text{cleaning}(A \times B \times K)$
In
  $\forall \xrightarrow{i/o^A o^B} \in T^{\tilde{P}}$ *// $\tilde{P}$'s transitions*
    $o^A = o^B$

4) Consumption: Now, let's focus on the consumption part of the property:

Let
  $\tilde{P} = \text{cleaning}(A \times B \times K)$
In
  $\forall t \in \tilde{P}$
    Let
      $(a_j b_j k_j)_{j=0}^{j=n} = States(\tilde{P}, t)$
    In
      $\sum_{j=0}^{n} C^A(a_j) \leq \sum_{j=0}^{n} C^B(b_j)$

The last line can be written this way: $\sum_{j=0}^{n}(C^B(b_j) - C^A(a_j)) \geq 0$.

Then we define a cost automaton $M^{\tilde{P}} = \langle \tilde{P}, C^{\tilde{P}} \rangle$ where the cost function $C^{\tilde{P}}$ associated with the product $\tilde{P}$ is defined as follows:

$$\forall a \in Q^A, \quad \forall b \in Q^B, \quad \forall k \in Q^K,$$

$$C^{\tilde{P}}(abk) = C^B(b) - C^A(a)$$

Now, we want to determine whether there is a trace which has a negative cost when executed on $\tilde{P}$. We want to explore all paths between the initial state and an accept state, focussing on the consumption. Note that we are using the non-functional hypothesis: if there were guards on transitions, some paths should not be explored.

For sake of simplicity, we construct the graph G induced by the cost automaton $M^{\tilde{P}}$ as defined in the previous section. The graph for our running example is given in Fig 2(b).

Notice that a path has the same cost (or weight) on the cost automata and on the induced graph. So we want to determine whether there is a path on G which has a negative weight. It is sufficient to find the weight of the lightest path. If it is positive, then all the other paths have a positive weight, then the consumption property is true. If it is strictly negative, then we have a negative path and consumption property is false. If there is no lightest path (this may happen when the graph has negative loops), there are paths of arbitrary light weight, then some of them are negative, then the consumption property is false.

Thus, for the consumption part, we have to check:

Let
  $G = \text{Graph}(\text{cleaning}(A \times B \times K))$
  wmin $= \text{weight-of-the-lightest-path}(G)$
In
  wmin is well defined and *wmin* ≥ 0

The work is not finished, because we did not detailed the procedure that finds the weight of the lightest path. There are many lightest-path algorithms but their behaviour is not defined when the graph has negative loops.

If there is a negative loop, it is necessarily reachable from the initial state and it can lead to the final state because the cost automaton was cleaned. Then, if some negative loop is detected, there is a negative path and the consumption property is false.

Finally, for the consumption part, we have to check:

Let
  $G = \text{Graph}(\text{cleaning}(A \times B \times K))$
  $\langle \textit{has-neg-loop}, \textit{wmin} \rangle = \text{lightest-path}(G)$
In
  not *has-neg-loop* and *wmin* ≥ 0

5) Discussion: This is in fact a decision procedure because both formula on outputs and energy are computable. Indeed, all the functions we use are computable: product and graph construction, cleaning and lightest-path finding. Moreover, visiting all the transitions of $\tilde{P}$ is feasible because the model is finite.

C. Complexity

We first focus on the product. In the worst case the states of a product are the Cartesian product of the states of its factors. However, in our case, we expect both component models to be close to each other because they represent the behaviour of the same component. Then the product of component models has about the same size as the detailed model. Moreover, we expect the model describing the context to be small (typically about 10 states) because it is a simple recogniser written by hand. Based on these two hypotheses we assume that the product of the three automata has almost the same size than the detailed model. Let compute the complexity relatively to its size.

The construction of the product has a linear cost, and so has the cleaning operation because it consists in visiting all states and transitions.

The procedure dedicated to the output property examines every transition, its cost is also linear.

The one dedicated to the consumption property first constructs a graph. Since the graph has almost the same structure as the product, its construction has a linear cost. Then a lightest-path algorithm is used on the graph. There are many such algorithms. To the best of our knowledge, the most efficient one that we can use in the presence of negative loops is the Ford-Bellman [5], [6] one. Its complexity is $O(n \times m)$ where n is the number of states and m the number of transitions. In the worst case there is a transition between any two states and $m = n^2$, but in our case, the number of transitions from a given state is bounded by the number of distinct inputs that can be received. This number is small compared to the number of states, then $m = O(n)$ and the complexity of the lightest-path algorithm is $O(n^2)$.

To summarise, the complexity of our decision procedure is the complexity of the Ford-Bellman algorithm, that is $O(n^2)$.

D. Counter-example generation

As abstract models are built by hand, it would be useful to get a counter-example of the abstraction relation in case it is false.

A counter-example consists of a trace in the context for which either the output property or the consumption property does not hold.

In case the abstraction is false because of outputs, the procedure detects a transition with two different outputs. A counter-example is a trace passing through this transition. It can be exhibited by finding a path from the initial state to this transition and another one from it to an accept state. Paths necessarily exist because the transition belongs to the cleaned product. These two searches are less expensive than the lightest-path algorithm and so do not change the complexity of the procedure.

In case all outputs are the same but energy is not over-approximated for some trace, we distinguish two cases.

If the graph has no negative loop, the Ford-Bellman algorithm exhibits a lightest path, and it is our counter-example.

Otherwise we would like to find one of the negative loops. All the lightest-path algorithms we found do not do that. That is why we slightly alter the Ford-Bellman algorithm to make it exhibit a simple negative loop. The change is simple. The standard Ford-Bellman algorithm iteratively looks for paths of size 1, 2, $\ldots n-1$ if n is the number of vertices, because in case there are no negative loop, there is a lightest path of size less or equal to $n-1$ between any two different vertices. To deal with negative loops, we have to look for paths of size n, because a negative loop could go through all n vertices once. In the n^{th} loop iteration, we remember the transition which was last used to find a lightest path. This transition is part of a negative cycle and the procedure usually used to find paths in the standard Ford-Bellman algorithm can be used here to find a negative loop, as we proved in [7]. Our extension does not alter the complexity of the standard algorithm.

Once we have a negative loop, we find two paths as previously, from the initial state to one state of the loop, and from it to a final state.

Finally, we can exhibit a counter-example without increasing the complexity.

V. Implementation

We implemented the decision procedure for the abstraction we proposed in Python [8].

We used the ARGOS language [9] to describe automata and the ARGOS compiler [10] to compute the synchronous product of these automata.

Another point is the choice of a lightest-path algorithm. We need an algorithm which finds the lightest path between two given vertices and allows for negative weights on edges and negative cycles. We choose Ford-Bellman algorithm because it has the smaller complexity. Moreover we managed to alter it so as to exhibit a negative loop in case there are some, as described in Section IV-D.

We tested our program with small examples and it worked in less than one second. We have no doubt that it will be quick for industrial problems, because our algorithm is efficient and the models we have to deal with are quite small. As an example, hardware component models (e.g. radio) have about 10 states.

VI. Conclusion

We have presented an automatic decision procedure for the abstraction relation which uses graph algorithms to check the over-consumption of the abstracted model against the detailed one. In case the abstraction is false, we provide a procedure to find a counter-example. It can be used to guarantee that an abstract model, smaller than the detailed one, satisfies the property of over-consumption. Then the validation algorithm can use the new model instead of the detailed one. It helps avoiding blow-up. The procedure is efficient so this gain is not balanced by an expensive pre-process.

Since Ludovic SAMPER proved the congruence of this abstraction relation [1], it is possible to use our procedure on component models. It is useful because it is much easier to find an abstract model of a small component than one of a full complex system.

This work can be applied to embedded systems, where energy consumption is a major issue. Besides, it will be used to study the worst-case lifespan of wireless sensor networks, as part of ARESA [11] project.

As a further work we plan to deal with functional properties in order to study systems whose behaviour changes according to the quantity of available energy. More complex models to represent components is a must. We plan to use automata with some kinds of guards allowed on the edges. The challenge is to find a tradeoff between the expressiveness of the models and the efficiency of methods we will apply on them. Concerning the decision procedure for the abstraction, we will first explore methods based on abstract interpretation [2]. They are too heavy for the data we use in this paper but suit the extensions we consider. Notice that since the composition of component

models with the context model seems unavoidable, the context model can be enriched as much as the component model without making the final working structure more complex.

REFERENCES

[1] L. Samper, "Modélisations et analyses de réseaux de capteurs," Ph.D. dissertation, INPG, 2008.

[2] P. Cousot and R. Cousot, "Static determination of dynamic properties of programs," in *Proceedings of the Second International Symposium on Programming*. Dunod, Paris, France, 1976, pp. 106–130.

[3] D. Cachera, T. Jensen, A. Jobin, and P. Sotin, "Long-run cost analysis by approximation of linear operators over dioids," *Lecture Notes in Computer Science*, vol. 5140, pp. 122–138, 2008.

[4] G. Behrmann, A. Fehnker, T. Hune, K. Larsen, P. Pettersson, J. Romijn, and F. Vaandrager, "Minimum-Cost Reachability for Priced Timed Automata," in *Proceedings of the 4th International Workshop on Hybrid Systems: Computation and Control*. Springer-Verlag London, UK, 2001, pp. 147–161.

[5] R. Bellman, "On a routing problem," *Quarterly of Applied Mathematics*, vol. 16, no. 1, pp. 87–90, 1958.

[6] L. R. Ford, Jr. and D. R. Fulkerson, *Flows in Networks*. Princeton University Press, 1962.

[7] L. Lugrin, "Using Abstraction for the Validation of Non-Functional Properties," Master's thesis, UJF, 2009.

[8] "Python," http://www.python.org/.

[9] F. Maraninchi and Y. Rémond, "Argos: an automaton-based synchronous language," *Computer Languages*, vol. 27, no. 1/3, pp. 61–92, 2001.

[10] D. Stauch, "Compilateur Argos," http://www-verimag.imag.fr/~altisen/DSTAUCH/ArgosCompiler/.

[11] "ARESA," http://aresa-project.insa-lyon.fr/.