Analyzing an Embedded Sensor with Timed Automata in Uppaal

Timothy Bourke, Arcot Sowmya

Analyzing an Embedded Sensor with Timed Automata in Uppaal

TIMOTHY BOURKE, INRIA and École normale supérieure
ARCOT SOWMYA, University of NSW

An infrared sensor is modeled and analyzed in Uppaal. The sensor typifies the sort of component that engineers regularly integrate into larger systems by writing interface hardware and software.

In all, three main models are developed. For the first, the timing diagram of the sensor is interpreted and modeled as a timed safety automaton. This model serves as a specification for the complete system. A second model that emphasizes the separate roles of driver and sensor is then developed. It is validated against the timing diagram model using an existing construction that permits the verification of timed trace inclusion, for certain models, by reachability analysis (i.e., model checking). A transmission correctness property is also stated by means of an auxiliary automaton and shown to be satisfied by the model.

A third model is created from an assembly language driver program, using a direct translation from the instruction set of a processor with simple timing behavior. This model is validated against the driver component of the second timing diagram model using the timed trace inclusion validation technique. The approach and its limitations offer insight into the nature and challenges of programming in real time.

Categories and Subject Descriptors: C.3 [**Special-purpose and Application-based Systems**]: Real-time and embedded systems; D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking*; *Validation*; *Formal methods*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Mechanical verification*; *Specification techniques*

General Terms: Design, Documentation, Verification

Additional Key Words and Phrases: Timed automata, Uppaal, Timing diagrams, Timed trace inclusion

ACM Reference Format:
Bourke, T., Sowmya, A. 2012. Analyzing an embedded sensor with Timed Automata in Uppaal ACM Trans. Embedd. Comput. Syst. 0, 0, Article 0 (2012), 25 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Integrating specialized components is fundamental to many embedded engineering projects. The behavior of these components is often specified in informal timing diagrams which are interpreted by engineers during the design of interface hardware and software. This article presents the modeling and analysis of one such component and its interface: an infrared sensor and its assembly language driver. The interest is in applying timed safety automata [Henzinger et al. 1994] and reachability analysis in Uppaal [Larsen et al. 1997] to a concrete example from practice, as opposed to an abstract algorithm or protocol. The example itself is novel and treated in detail.

The sensor is a small-scale case study: its specification sheet has only four pages and a driver can be written in about twenty lines of assembly language. It suits the intent

---

The work described was performed while the first author was a student at UNSW and NICTA.

Author's addresses: T. Bourke, Département d'Informatique, École normale supérieure, Paris; A. Sowmya, School of Computer Science and Engineering, University of NSW, Sydney.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

of treating an example taken from practice rather than one contrived or adjusted to exhibit points of theoretical interest. The sensor and the timing diagram in its data sheet are effectively treated as artifacts of intrinsic interest to be observed 'in the wild'. This is a reasonable view, even though both are man-made, for two reasons. First, because the idea, initially anyway, is to observe, rather than to change, practice. And, second, because engineers must often work from specifications and experiment alone without recourse to component designers. The choice of a single, small, and arbitrary example, however, certainly limits generalization: some characteristics of the device may be idiosyncratic, and other interesting characteristics are certainly not represented. At the very least, the case study is a data point to be added to the set of existing examples.

Quantitative time is integral to understanding and directing the sensor. Mostly because its timing diagram prescribes a way for two asynchronous components to communicate with minimal interconnections: two wires in fact. Both the description of these timing constraints and the way they may be met by implementations are examined. Although the meaning of most signal changes depends more on protocol state than on precise time of occurrence, an open-loop version of the driver is also considered; its operation relies on the passage of time rather than the receipt of events.

The sensor and its timing diagram are described in §2. A careful interpretation of the timing diagram, and the timed automaton model that results are presented in §3. The timed automaton model is then refined into a *split model* of cooperating timed automata where the separate roles and the interaction of the sensor and driver are emphasized. This model is presented in §4, which also contains a description of two analyzes in Uppaal. In the first, a testing automaton is constructed from the original timing diagram model and analyzed together with the split model to show timed trace inclusion. In the second, a transmission correctness property is formulated and verified by reachability analysis. The driver component of the split model serves as a specification in §5, where an assembly language driver is modeled and validated against the specification via timed trace inclusion testing. The article concludes in §6 with a discussion of the results and some suggestions for extending the research.

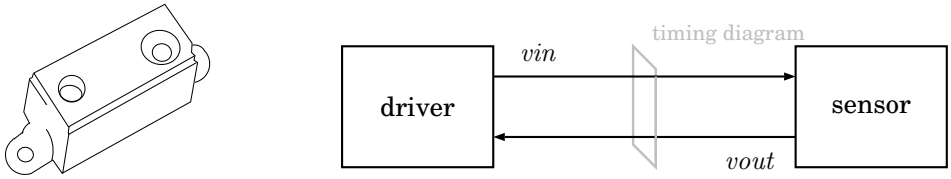
2. THE SHARP GP2D02 RANGE SENSOR

The Sharp GP2D02 range sensor exemplifies the sort of component which engineers integrate into embedded systems. The sensor is an electro-optical component that exploits physical principles to produce results that are useful within a larger system.

The choice of investigating the GP2D02 is somewhat arbitrary. Significantly, however, the device specification is relatively simple, includes timing constraints, and mixes event-driven responses with sampling. An awareness of the example's strengths and weaknesses could guide future experiments. The philosophy is to examine realistic examples for opportunities where they may be better understood or developed, rather than to choose examples which suit a particular development approach.

2.1. Overview

The sensor, refer Figure 1a, is a small ($14 \times 29 \times 14\mathrm{mm}$) box. Visible features include two mounting slots, an infrared light emitting diode, a lens that covers a detecting surface, and four electrical terminals: voltage, ground, input (vin), and output ($vout$). The sensor measures the distance to an object by emitting infrared beams from the diode which are then reflected by the object. The distance is estimated by measuring the position of the reflected beams along the detecting surface. Measurement cycles are triggered and 8-bit distance estimates read from the output terminal by providing a suitable signal on the input terminal. This signal and other details are described in a data sheet [Sharp Corporation 1997]. The behavioral descriptions are informal and subject to interpretation against a background of engineering practice.



(a) Physical appearance (b) Utilization

Fig. 1: The GP2D02 Sensor.

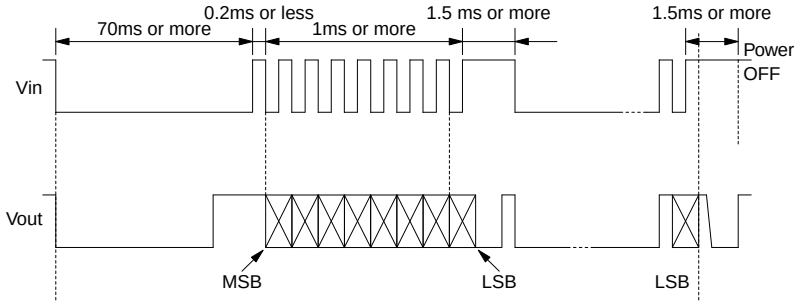


Fig. 2: The sensor timing diagram [Sharp Corporation 1997].

In applications, a sensor is connected to another device, here termed the *driver*, as shown in Figure 1b. The driver controls the signal level on vin, and the sensor controls the level on $vout$. Three interrelated models of this system are presented in this article. The first, in §3, describes the combined protocol—the causal relations and timing constraints between voltage levels and voltage level changes on the two wires—as represented by a timing diagram.The sensor is treated as a black box that guarantees the stated behaviors on $vout$ provided the driver conforms to the expected behaviors on vin. The individual roles of driver and sensor are emphasized in the second model, presented in §4. The third model, presented in §5, concerns a driver implementation.

2.2. Timing Diagram

The sensor timing diagram is reproduced in Figure 2. This figure is copied directly from the sensor data sheet (with one minor change: the lower signal, here named *vout*, is referred to as *output* in the original). The upper signal vin specifies the input sequences that may be applied to the unit. The lower signal $vout$ specifies the expected response.

The first falling edge on vin triggers a range reading which may take at most 70ms. A series of pulses are then applied to clock data out of the sensor, and, finally, at least 1.5ms must elapse before either repeating or terminating the process.

The choice between stopping and continuing is the only point where control behaviors, as opposed to timing or data valuations, branch. Both possible scenarios are shown at right in the diagram; in each a pulse on vin is labeled with *1.5ms or more*. The first pulse on vin is interpreted as triggering a new range reading. The second, labeled with *Power OFF*, is interpreted as the decision to switch the sensor off. Between these alternatives, each signal line is broken by a dashed horizontal section. Clearly, representing branching in a timing diagram is awkward.

There are four dashed vertical lines between the two signal waveforms. These seem to indicate synchronizing events, from left: commencing a range reading, triggering a change in $vout$ for the most significant bit (MSB), similarly for the least significant

bit (LSB), and returning to a low $vout$ level after an unspecified, though bounded, delay. There is a fifth vertical line that does not reach the lower waveform. Its meaning is difficult to decipher, it seems to imply that $vout$ will become high again within 1.5ms.

The diagram indicates that vin should remain constant for at least 70ms after the falling transition that triggers a range reading, during which time the $vout$ signal will change. Some implementations [Griebling 1999; Ramsey 2001] ignore the timing constraint and act instead as soon as $vout$ becomes high—this possibility is modeled.

The crossed boxes on the $vout$ signal, between most and least significant bits, indicate data non-determinism. The signal value, either high or low, depends on the range reading and must typically be sampled since there will not necessarily be a detectable event. For accurate sampling, it is necessary to know bounds on when, relative to other signal events, the $vout$ level will be stable. The timing diagram could be more explicit, but it seems that changes in $vout$ are usually triggered by falling transitions on vin. The exact behavior, however, of $vout$ after the least-significant bit has been sampled is not clear. If the last bit is zero, $vout$ must return to the high level before 1.5ms elapses. If it is one, it seems that $vout$ must go to a low level first before returning to a high level. An engineer could clarify such unclear details, should they prove important, by running experiments with an instance of the device. This is an effective approach, but, at least in principle, such observed behaviors may change between different versions of the specified device. It is assumed that a rising vin transition after the last sample triggers $vout$ to fall if necessary and then rise again.

According to the diagram, the sensor requires at least 1.5ms between the end of one complete range reading and the beginning of the next. During this period there is a rising transition on $vout$ which could indicate that the sensor is ready to make another reading before the whole 1.5ms has elapsed. The specification is not clear.

The *70ms or more* and *1.5ms or more* constraints are readily justifiable: it takes time for the device to make measurements and to recover; less so the *0.2ms or less* constraint. Rather than seek a motive, the constraints will simply be accepted as given. Also, rather than interpret *0.2ms or less* as a constraint for all the other positive pulses ⎍ and perhaps also the negative pulses ⎏, it is assumed to pertain only to the first.

The pulses, though, must definitely have some minimum value, as suggested by the *1ms or more* constraint. The minimum width of a positive pulse will be represented by minmark, and that of a negative pulse by minspace. The *1ms or more* constraint is not otherwise further interpreted. The values of minmark and minspace will depend on the sensor's (unspecified) internal electronics and properties of the interconnection (such as wire capacitance). They are assumed to be equal to zero from now on.

Although not perfect, the timing diagram is adequate for interfacing with the sensor once its ambiguities have been resolved. Converting such an informal description into a precise notation quickly reveals what is clear and what is not.

3. TIMING DIAGRAM MODEL

In this section, a timed automaton model of the timing diagram is described. The sensor timing diagram is ordinarily read as the specification for a device driver—a circuit or program for triggering the sensor and extracting a reading. But it could also be taken as a template for creating different types of compatible sensors. In either case, one would classify events and constraints within the diagram from the perspective of one side or the other, as inputs or outputs, or as assumptions or guarantees. In this section sensor is not distinguished from driver. The focus is instead on the timing diagram as an artifact in itself. It is modeled as a timed automaton, thus providing a precise interpretation. Alternative modeling choices and possible variations are discussed.

There are five subsections. In §3.1, some benefits of producing such a formal model and some of the philosophies that guide its creation are discussed. The choice of al-

phabet for the model is described in §3.2. A detailed description of the timing diagram model is presented in §3.3. Liveness requirements are discussed in §3.4. Lastly, in §3.5, some limitations of the case study are listed.

3.1. Rationale and guiding philosophy

Timing diagrams are usually understood through convention, culture, and practical experimentation rather than in formal terms, despite several proposed formalisms.[1] There are at least three reasons for modeling a timing diagram in a formal framework:

(1) Detailed questions are asked of the specification. Its meaning is clarified and ambiguities or omissions may be discovered and noted.
(2) A formal specification defines a notion of correctness against which other artifacts, such as sensor and driver implementations, may be validated.
(3) Tools for validating and transforming some types of models exist, synthesis being a special case of transformation into an executable form.

Implementation in a specific programming language is effectively a translation to a formal, or at least semi-formal, notation. There are, however, important differences. While implementation also requires attention to technical details, certain language features may behave differently across compilers and platforms. Significantly, properly expressing timing details may be difficult. A program is, at best, a single *reference implementation*, it cannot usually reflect the full range of permitted behaviors. Furthermore, there may be confusion between which features of the program are properties of the object under consideration and which are necessitated by the chosen language.

The key difference between a model and an implementation is one of abstraction. Models will usually ignore details essential to implementations, and conversely, implementations will typically be too constraining to act as models for all purposes. When specifying timing, and other behaviors, it is particularly desirable to ignore distracting implementation details.[2] Ideally though, the models and implementations of a system are interrelated in a precise manner.

The timing diagram, Figure 2, defines a partial ordering and relative timing constraints on a set of events. The aim is to capture precisely this information, and no more, in a timed automaton model. Justification will be offered for all compromises.

3.2. Choosing an alphabet

The transitions from one signal level to another are of most importance in this particular timing diagram. They are the main events that the model must address.

The vin and $vout$ signals can be described in terms of *edges* and *levels*. Edges are characterized by direction of change, rising or falling; ideally they are instantaneous with a definite time of occurrence. Levels are characterized by a state, high or low, that persists over an interval between two edges. The sensor timing diagram depicts ideal signal edges. Other timing diagrams acknowledge the non-instantaneous nature of edges by using vertical lines with a slight slant.

There are at least three different ways to associate events with transitions. The first associates a distinct event with every change in the value of a signal, and each event has a unique label: for example, $e_1, e_2, \ldots, e_n$. Such detail is tedious and, for the timing diagram model, unnecessary. The second way associates an action with each signal, for example vin and vout. Different occurrences of the actions are distinguished by relative order of occurrence. The third way is similar, it associates two actions with each signal:

[1] Bourke [2009, Appendix F] presents the model in a formal timing diagram notation and cites related work.
[2] Although, conversely, most real-time programming is characterized by careful attention to such detail.

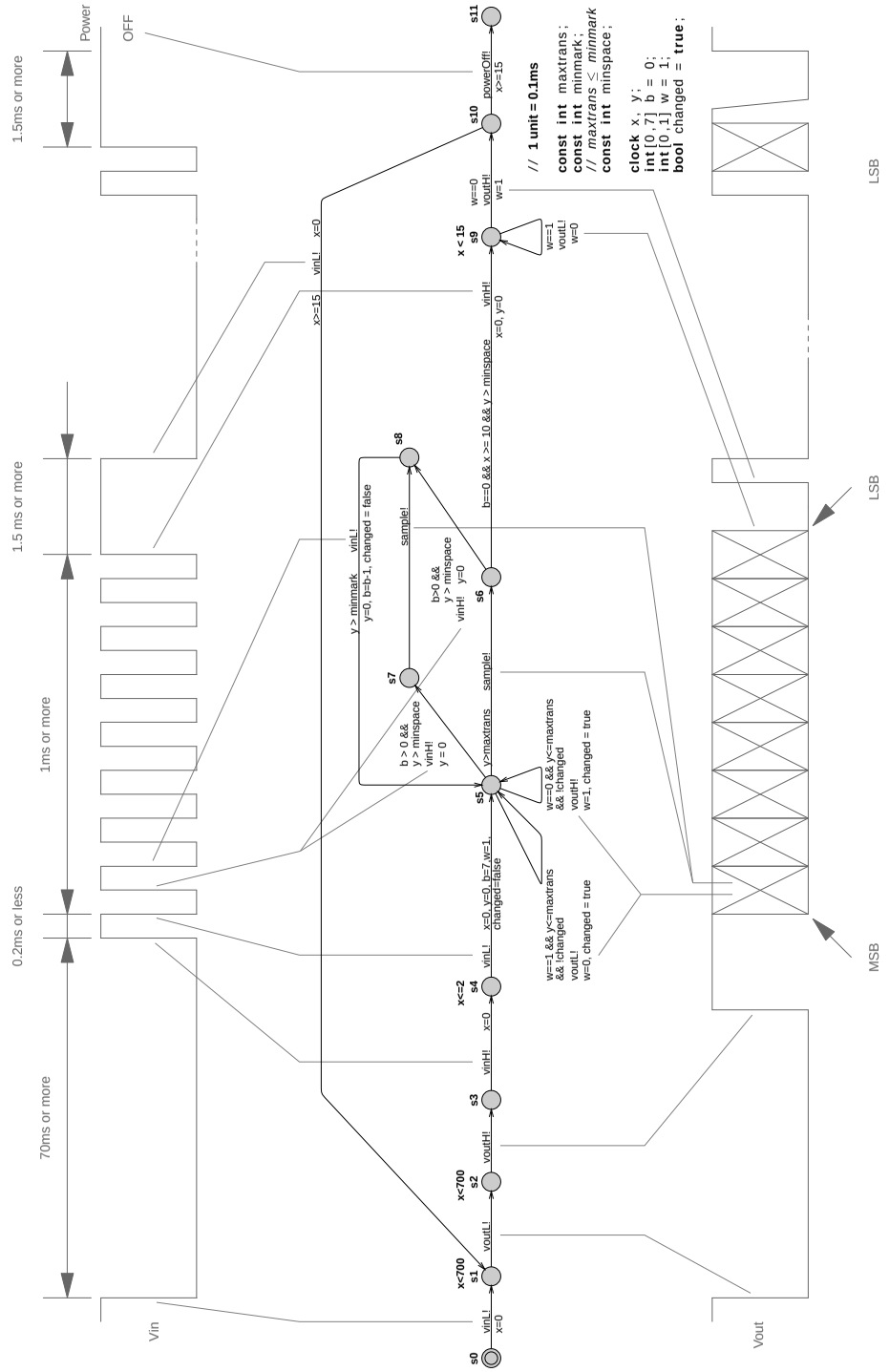
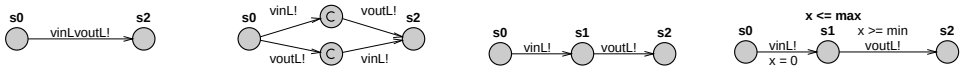


Fig. 3: The timing diagram model (compared to the timing diagram, in gray).



(a) composite (b) interleaved (c) causal (d) bounded causal

Fig. 4: Some alternatives for modeling two related events.

one for a rising transition on the signal, the other for a falling transition. This extra detail is not absolutely necessary, since rising and falling actions must alternate strictly, but it clarifies the relationship between the model and the timing diagram, making both easier to think about and to describe. This third choice is thus adopted. Transitions labeled vinL and vinH correspond to falling and rising transitions, respectively, on vin. Likewise for voutL and voutH on $vout$.

The rising and falling transitions on vin and $vout$ are explicit in the timing diagram, but the diagram also constrains two other events: powering off and sampling.

A powerOff action is introduced to represent the act of turning the sensor off.

The timing diagram is not explicit about when the level on $vout$ can be sampled, even though it is an important feature of the protocol. The timing diagram model will be more precise: a sample action is introduced to represent instants when readings can be accurately taken from the $vout$ line. This new action is controversial since it is not explicit in the timing diagram. But, arguably, it would be inferred by engineers anyway, and should perhaps have been included.

3.3. An explanation of the model

The model in Figure 3 is the result of many gradual refinements. It is thought to be an accurate interpretation of the timing diagram of Figure 2, which is reproduced in gray to aid comparison.[3]

The model is expressed as an Uppaal (4.0.13) timed automaton so that its relationships with other models can be verified automatically. As a consequence, transitions must be labeled as inputs or outputs despite the neutral stance taken on the issue. They have all been made outputs, simply because this makes validation in §4.2 easier. The model is to be interpreted as an open system, one that defines a set of timed sequences of allowed actions, rather than as part of a closed system that can only act when another component is willing to synchronize.

In a valid Uppaal model all timing constants must be integers: in the model, one unit of time is equivalent to 0.1ms in the timing diagram.

There are three main phases in the timing diagram protocol: the initial triggering of a range reading (§3.3.1), transferring the resulting value bit-by-bit (§3.3.2), and finally deciding whether to power off or to repeat the process (§3.3.3).

3.3.1. Initial triggering. A range reading cycle is triggered from the initial location, s_0, by a falling transition on vin: vinL. According to the timing diagram voutL occurs simultaneously. But, conceptually, vinL causes voutL. There are at least four ways to model the relationship between the two events:

(1) A single event composed of two others: vinL·voutL. Such a composite event results from the synchronization of two components, but in this setting the event would later be decomposed into separate actions of the driver and sensor. There are two obvious expressions in Uppaal. The one shown in Figure 4a is direct but makes decomposition clumsy. The one shown in Figure 4b tries to capture the commutativity and atomicity of

[3]A small improvement could perhaps be made. The model could be adjusted to allow termination without taking any range-readings by adding an extra transition from s_0 to s_{11} labeled with powerOff.

the event combination operator, though atomicity is not guaranteed by the committed locations,[4] marked Ⓒ, in models where other such locations occur concurrently.

(2) One event after the other, capturing the causal dependency but not the implied synchronization, as in Figure 4c. This choice is adopted for the timing diagram model.

(3) One event following the other but with a clock reset on entry, $x = 0$, and an invariant on the middle location, $x \leq \mathsf{max}$, to express necessity of occurrence; as in Figure 4d. When max is 0, the events are simultaneous but ordered, as for the *micro steps* of state-diagram languages, or the *delta steps* of discrete-event languages. Adding a lower bound, $x \geq \mathsf{min}$, gives a model that more accurately reflects physical reality.

The *70ms or more* constraint is given between initial vinL and vinH events, but causal dependencies between vinL and voutL, and voutL and voutH, and voutH and vinH effectively also constrain the intermediate voutL and voutH events. The first of the causal dependencies has just been discussed. The second arises because signal levels must alternate. The third is less explicit, it is divined from some extra knowledge of the sensor, which, when it has made a reading, raises the output signal. Thus both locations s_1 and s_2 have an invariant label $x < 700$, which guarantees the occurrence of voutL and voutH before $70\mathrm{ms}$ passes. Strict compliance with the timing diagram would require an $x \geq 700$ guard on the vinH transition between s_3 and s_4. The model varies on this point: the driver may proceed as soon as voutH is detected—as earlier discussed.

The vinH event must be followed by vinL within *0.2ms or less*. This constraint is modeled by resetting x when the former event occurs and adding an $x \leq 2$ invariant to s_4. The vinL transition sets the clocks x and y, and the variables b, w, and changed:

x	records the time elapsed since the start of transmission of the last MSB.
y	records the time elapsed since the last change of vin
b	counts down eight transmitted bits through locations s_5, s_6, s_7, and s_8,
w	tracks $vout$ to ensure strict alternation of voutL and voutH events,
$changed$	ensures that at most one output event occurs for each sampled bit.

3.3.2. Data transfer. A transmission cycle begins after a falling edge on vin, vinL, prompts the sensor to transmit the next bit of the range reading. Both the initial vinL, from s_4, and the looping vinL, from s_8, lead to s_5.

The level of $vout$ is only allowed to change while s_5 is active and within maxtrans of the triggering vinL. The self-loops on s_5 express possible changes on $vout$; they are discussed in more detail later. The timing constraint is measured by the clock y which is reset, within the loop, whenever there is a change on vin. The constant maxtrans combines an assumption on the maximum time the sensor will take to change the vout level after being triggered by vinL, and the time required to transmit any change through the wiring and interface electronics. It is not explicit in the timing diagram but the sampling period is not well defined without it.

From s_5, the driver must sample the $vout$ level and then return vin to a high level. Both actions, sample and vinH, must happen after the maxtrans delay and before the next vinL. They are causally-independent for all but the LSB, that is while $\mathsf{b} > 0$. Such relationships are naturally modeled with parallelism; but expressing nested parallelism in Uppaal is awkward and requires additional synchronizations. Since there are only two actions all of their possible interleavings can be explicitly modeled: the path s_5–s_6–s_8 samples first and then raises vin, and the path s_5–s_7–s_8 raises vin first and then samples. But such an approach quickly becomes untenable as the number of mutually-independent actions increases. For the case of the LSB, when $\mathsf{b} = 0$, only

[4]Transitions from committed locations have priority over those from non-committed locations and delays.

one of the interleavings is allowed. The sample action occurs first because it is assumed that vinH signals to the sensor that sampling is complete.

The alternation of vinL and vinH actions within the transmission loop gives alternating negative and positive pulses on vin. Both types of pulse have a minimum width: minspace for the negative pulses and minmark for the positive ones. This is expressed in the timed automaton by guard expressions on clock y. The timing diagram is not explicit about minimum pulse widths. It states only that the eight negative pulses and seven positive pulses that comprise each cycle must take *1ms or more*. Rather than assume minmark $=$ minspace $= 1/15\text{ms}$, the model is validated with minmark $=$ minspace $= 0$; the most permissive choice. The *1ms or more* lower bound is enforced separately by the $x > 10$ guard on the vinH transition leaving the loop.

The model assumes that maxtrans $\leq$minspace, but rather than add an extra clause (... && y > maxtrans) to the guards on edges s_5–s_7 and s_6–s_8, this constraint on the constants is stated outside the model. Setting minmark $= 0$ thus implies that maxtrans $= 0$. An alternative approach would be to duplicate the voutL and voutH self-loops on s_7 and add the guard y > maxtrans to the transition between s_7 and s_8; but it seems less natural to allow the driver to act before the sensor value has stabilized.

In the timing diagram, Figure 2, there are several crossed boxes in the $vout$ signal between MSB and LSB annotations. They represent data non-determinism; the signal may change or remain constant from one bit to the next depending on the value being transmitted. The crossed boxes thus abstract over 2^8 possible data signals, not distinguishing variations in timing. There are at least five ways to model them:

(1) By not modeling them explicitly. This results in a simpler model because there are fewer transitions and the behavior of $vout$ after the LSB is easier to model. Whilst sufficient, from the driver's perspective, to only specify when sampling may occur, the timing diagram model tries to avoid bias toward either role.

(2) By marking the event with a change label, thereby ignoring the specific value. This, however, would complicate later verifications of data transmission. It is also misleading because the level of $vout$ will not change if adjacent bits are identical.

(3) By non-deterministic choice between both possible transitions. This technique refines the change action into two different actions: rising and falling transitions. It is more in the spirit of the timing diagram, as a partially ordered set of transition events, but it incorrectly infers that an update of the output level is always observable, which depends, rather, on the values of adjacent bits in a data reading.

(4) By modeling both types of transition, but also including the level of $vout$ in the automaton state. This treats the observability of events more accurately—no event occurs if the level does not change—and also facilitates the verification of data transmission (§4.3). This technique was chosen for the timing diagram model of Figure 3. Variable w encodes the level status and variable changed ensures that at most one event occurs per data bit. In principle, one could write an event-triggered driver that responds to the presence or absence, in a given period, of transitions on $vout$.

(5) By modeling the possibility that $vout$ may change several times before settling, within maxtrans units, to a constant value. This approach [Vaandrager and de Groot 2006] is closer to physical reality where a signal may, after a change in level, oscillate unpredictably before stabilizing. In this situation a driver triggered by events, rather than one that samples the signal level, is impractical. The timing diagram model can be adjusted to use this technique by removing all references to the changed variable.

After eight transmission cycles, when b is zero, a transition leaves the loop from s_6 on vinH. If the last $vout$ level was high (the least significant bit (LSB) was one; w $= 1$), it must now be set to a low level via a voutL event. The timing diagram is not precise about when this event occurs, but it must precede the voutH event, which in turn must

precede the vinH event that exits the sampling loop within 1.5ms. Thus both the voutL self-loop on s_9 and the voutH transition to s_{10} are constrained by the invariant $x < 15$.

3.3.3. Power off or repeat. After triggering a range reading and receiving the result, there is a choice of terminating, the powerOff action to s_{11}, or of requesting another reading, the vinL action back to s_1. Neither can happen until the sensor is ready. The timing diagram demands that the choice be made *1.5ms or more* after the vinH event that ends sampling. This constraint is expressed in the model by $x \geq 15$ guards on transitions from s_{10}. An alternative is to interpret a voutH within this period as an indication that the sensor is ready; as when waiting for completion of a range reading.

3.4. Liveness and progress

The location invariants in the timing diagram model specify when actions are necessary; or equivalently, when unbounded delay is forbidden. Where there is no invariant the protocol may stop completely. The vinL, vinH, sample, and powerOff actions from s_0, s_3, s_6, s_7, s_8, and s_{10} need never occur. Similarly for s_5 where voutL and voutH may possibly, but need not, occur within bounded time, and s_{11} where unbounded delay is mandatory.

The timing diagram contains bounded liveness guarantees on all sensor responses. These upper timing bounds are modeled as location invariants, on s_1, s_2, and s_4, or as transition guards, on the loops at s_5. The sensor must always respond to the driver within a fixed period of time. A time-bounded response from the driver is only expected at s_4—and even this *0.2ms or less* constraint is somewhat dubious.

It is sometimes useful to specify the necessity of progress without stating an explicit time bound. Such abstract liveness properties can be expressed via an acceptance criterion, like Büchi or Muller conditions, and a subset of accepting states. Using Büchi conditions, various liveness constraints could be added to the timing diagram model:

— *Require complete protocol cycles*: make both s_{10} and s_{11} accepting and add a τ-transition[5] self-loop to s_{11}. Range reading cycles must then run to completion—reading taken and transmitted—once a triggering vinL has occurred. Cycles may be triggered continually forever, or a finite number of times until powerOff occurs.

— *Allow a finite number of complete readings without mandating* powerOff : make s_0, s_{10}, and s_{11} accepting and add τ-transition self-loops to all three. Cycles must then run to completion once triggered, but neither endless cycles nor eventual powerOff are required: nothing need occur initially or between readings.

There are a few ways of stating in Uppaal that a model does not delay indefinitely: by giving a specific upper bound in a location safety invariant or equivalently using an urgent location, committed location, or urgent channels, though these are less appropriate in an open model since their behavior in composition is awkward. But Uppaal cannot model abstract liveness requirements.

3.5. Limitations

At least seven factors limit generalizations from this example:

(1) The timing diagram represents communications between a driver and sensor where the former is essentially master and interaction is limited. In particular, it is possible for the driver to operate without any direct feedback from the sensor.

(2) Although the communications involve two components running concurrently, the protocol is essentially sequential. That is, neither component performs many independent actions between synchronizations with the other.

[5]Such transitions are not labeled with an action and thus cannot synchronize with other transitions.

(3) There is almost no branching in the control structure. In particular, when one component waits for the other, it always expects a specific signal—its actions do not depend on which signal occurs, they are only delayed until the awaited signal.

(4) The driver and sensor essentially form a closed system. That the environment affects transmitted infrared beams has no bearing on the design.

(5) Data values do not significantly affect the behavior of either component.

(6) Timing constraints are stated solely between driver events, but never between sensor and driver events, or between two sensor events, and although there are sometimes implications for sensor events no constraints are stated directly. The constraints do not overlap, nor do they depend on the particular values of earlier delays.

(7) Component behavior does not depend on the measured length of delays.

Limitations 5 and 7 could be overcome, but only by considering an unusual driver that would detect event occurrence and absence rather than sample signal levels.

4. DRIVER/SENSOR SPLIT MODEL

While the individual roles of driver and sensor were largely ignored in the timing diagram model they are the focus of §4.1, where a *split model*, which effectively adds behavioral detail to the structural diagram of Figure 1b, is presented. The relationship of this model to the timing diagram model is shown in §4.2. Besides corroborating the timing diagram model, there are two other advantages to constructing the split model. The first, discussed in §4.3, is that it can be used to verify a transmission correctness property. The second is explored in the following section, where the driver component of the split model becomes a specification for an assembly language implementation.

4.1. The split model

The first step in deriving driver and sensor components from the timing diagram model is to decide which actions each will control as outputs: An action is an output at a component if its occurrence is determined by that component. An action is an input at a component if its occurrence may influence that component's behavior. Actions vinL and vinH are sensor inputs and driver outputs, and actions voutL and voutH are sensor outputs and driver inputs. These assignments reflect both the reality of connections between driver and sensor and a desired division of responsibility. The events sample and powerOff are slightly different since they do not influence the behavior of the sensor: they are driver outputs but not sensor inputs.

In the interpretation of the timing diagram, in §3, the sensor only synchronizes with the driver at the triggering vinL, at the vinL for reading each bit, and at vinH after the LSB. These synchronizations are inferred from background knowledge about the sensor, although much is already determined by the predominantly sequential behavior of the protocol and the strict alternation of falling and rising transitions on a signal. The driver, on the other hand, need not synchronize at all with the sensor since the timing guarantees make open-loop control possible. Alternatively, it may synchronize on the voutH that indicates a completed reading. Both possibilities will be modeled.

4.1.1. Construction of the model. In the split model, the driver alone is supposed to be solely responsible for determining when events on vin occur, and similarly for the sensor on $vout$: output actions may not be constrained by other components. There are two ways to model this in Uppaal: either with handshake communication and the addition of input-enabling self-loops or with broadcast communication.

Rather than directly declaring allowed actions, Uppaal models contain channel declarations, each of which implies a specific pair of input and output actions. Communication on *standard channels* (declared with **chan**) requires the participation of two processes: an output action from a component is blocked when no other component

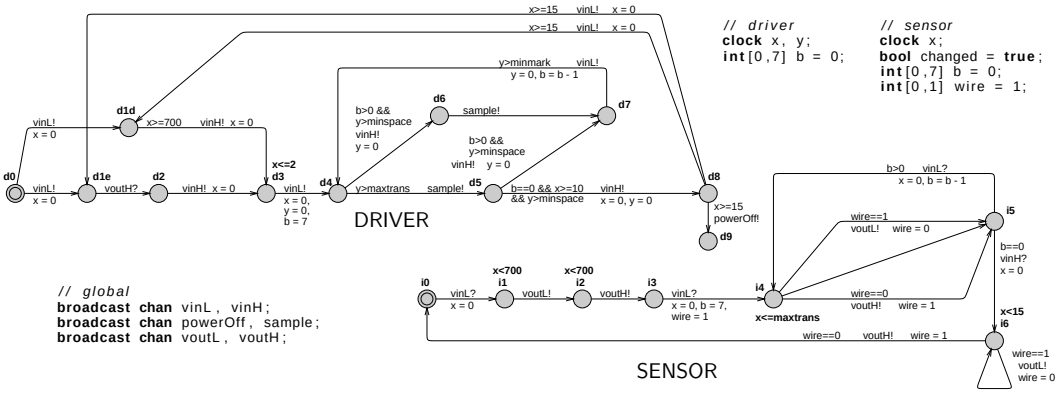


Fig. 5: The split model of the sensor timing diagram.

is willing to accept it by performing the corresponding input action. Ensuring input-enabledness with such channels is possible but requires complicating models with additional transitions; an approach that is briefly discussed in §4.1.3.

Uppaal also allows *broadcast channels* (declared with **broadcast chan**), where an output action may always occur and will be synchronized with a single, corresponding input action in each process where any such action is enabled. When using broadcast channels, no additional effort is required to ensure input-enabledness. A split model based on broadcast communication is shown in Figure 5. The model comprises two components in parallel:

$$\text{DRIVER} \parallel \text{SENSOR}.$$

The driver model was derived from the timing diagram model. All transitions on voutL were removed, as were all but the first on voutH, which was made an input. The variables wire and changed were removed, as were the $x < 700$ location invariants, and the initial sequence was duplicated through $\mathsf{d}_{1\mathsf{d}}$ with a new $x \geq 700$ guard.

The sensor model was derived in a similar way. All transitions on vinH but the one to s_9, whose guard was changed, were removed. All transitions on sample and powerOff were removed. The variable w was renamed to wire to avoid confusion. The variable changed was removed; its effect is represented in the sensor control structure by states i_4 (changed $= 0$) and i_5 (changed $= 1$). The voutL and voutH self-loops on s_5 become transitions between the two states, and a τ-transition is added for the case where there is no change. The purpose of clock x is changed, it appears in invariants on i_4 and i_6 that force the model to meet protocol timing assumptions.

4.1.2. Behavior of the model. The driver initiates a range-reading from d_0 by emitting a vinL output, on which the sensor synchronizes from i_0. The driver non-deterministically enters either $\mathsf{d}_{1\mathsf{d}}$, where it waits for *70ms or more* before continuing, or $\mathsf{d}_{1\mathsf{e}}$, where it can continue as soon as the sensor emits voutH. A similar arrangement exists from d_8. The choice could be modeled using τ-transitions, a committed location, and an extra state which would avoid duplicating the vinL transitions. But such a model would complicate time trace inclusion testing, where τ-transitions are forbidden. It can be argued that it is unreasonable to allow the driver to choose a different behavior at each iteration; the advantage of this model is that it can be specialized to one type of driver or the other by removing transitions.

In the original model, under the assumption that the sensor will respond to vinL with voutH within 70ms, the trace fragments accepted by the top path (d_0/d_8)–$\mathsf{d}_{1\mathsf{d}}$–d_3 are also

accepted by the bottom path (d_0/d_8)–d_{1e}–d_2–d_3. But both paths are included to make the driver model self-contained when used in isolation from the sensor model and to make implementation choices explicit. Furthermore, either branch of the choice can be eliminated to produce a more specific specification.

The driver sampling loop d_4–(d_5/d_6)–d_7–(d_4/d_8) is taken directly from the timing diagram model. The sensor sampling loop is simpler. After receiving a vinL input, the sensor acts within maxtrans units to transition between i_4 and i_5, either signaling a level change by synchronizing on voutL or voutH, depending on the local variable wire, or by not signaling any change, but instead changing state with a τ-transition.

Location invariants are present at d_3, i_1, i_2, i_4, and i_6. From each of these locations, there is at least one outgoing transition labeled with an output action. This is obligatory in the split model because otherwise one component could influence the behavior of another by 'stopping' time, either indefinitely, giving a Zeno trace, or until an awaited input action is forced to occur.

Finally, the driver component has, from d_8, the choice of delaying indefinitely, triggering another range reading, or powering the sensor off.

4.1.3. Alternative model with standard channels. Bourke [2009, Figures 4.20 and 4.21] shows that it is also possible to build a split model using handshake actions. Basically, the automata of Figure 5 are made input-enabled by augmenting them with extra loops—for voutL?, voutH? in DRIVER and for vinL? and vinH? in SENSOR—for every location and valuation (using guards) where such actions are not possible. An additional process is added to ensure that sample and powerOff outputs are never refused. All of the verifications described in the remaining sections can be adapted for this model.

While it can be argued that handshake communications are easier to think about than broadcast communications, since only two processes are ever involved in any synchronization, this is less convincing for models that involve two, or at most three, automata. Moreover, in this case there are no real obstacles to applying the testing construction for timed trace inclusion, which is usually only defined for handshake channels. All of the self-loop transitions needed when using handshake channels are annoying to maintain and distracting. Such problems only become worse in a model as the number of inputs and processes increase.

4.2. Verifying implementation

Both the split model and the timing diagram model are proposed as formalizations of the reading and transmission protocol. It is possible to show that the split model implements, in a precise sense, the protocol described by the timing diagram model.

4.2.1. The choice of relation. There are many ways to relate transition systems in general and timed automata in particular. The approach in this article follows that of Kaynar et al. [2006] in adopting timed trace inclusion.

Timed trace inclusion is a relatively simple notion of implementation. One timed automaton implements another timed automaton if the set of all timed traces of the former are a subset of the set of all timed traces of the latter—any safety properties proved of a specification model are immediately also true of its implementations. To claim that the split model implements the timing diagram model is to claim that any timed trace of the split model is also a valid timed trace of the timing diagram model. There are, however, some technicalities, which are discussed soon, because the split model is a network of timed automata rather than a single timed automaton.

Although timed trace inclusion is undecidable in general [Alur and Dill 1994], when the specification model is deterministic, which also effectively means free of τ-transitions, there are constructions for deciding it via reachability analysis in Uppaal [Jensen et al. 2000; Stoelinga 2002]. The basic idea is outlined in §4.2.2.

While timed trace inclusion allows safety properties to be inferred, deadlock and liveness are ignored. This fact is less limiting for timed models than for untimed models, because bounded liveness, the necessity of action within a fixed finite time, is a safety property. But this caveat must, nevertheless, be kept in mind. The separate verification of liveness and deadlock properties, including timed deadlock, is sufficient for the models of this article, since their branching structures are relatively uncomplicated.

Timed trace inclusion is an asymmetric relation. Using it to verify that the split model implements the timing diagram model only shows that the former never exceeds the behaviors permitted by the latter. Showing the same relation in the other direction would imply the stronger relation of timed trace equivalence of the models, in particular, that no valid behaviors are excluded from the split model. The models would be interchangeable modulo the limitations of timed trace equivalence. But, the testing construction used in this article only applies to single automata, not networks of automata. In recent work [Bourke et al. 2011], a technique based on timed games and implemented directly in the model-checking engine, has been applied to verify that the timing diagram model implements an adjusted version of the split model. But the present approach, using an unmodified model-checker, still has advantages in terms of transparency and relative simplicity, and working in one direction is sufficient to verify a driver implementation against the timing diagram.

4.2.2. Verification. The procedure for testing timed trace inclusion[6] within a specification, namely the timing diagram, has two steps: transforming the specification into a testing automaton, then performing reachability analysis of the result in parallel with the implementation automaton, that is, with the split model.

Several changes are necessary to transform a specification automaton into a testing automaton. A new location, called Err, is added. Inputs are made outputs and vice versa. Then from each location new transitions to Err are added for every action that cannot occur from that location. Input and output actions are considered as distinct even when on the same channel. Whether an action can occur or not may depend on the values of clocks and variables, so the guards of outgoing transitions must be considered. Furthermore, each location invariant is replaced by a τ-transition to Err that has the negated invariant as a guard. The result is a testing automaton that can 'observe' another automaton by running in parallel with it and synchronizing on each of its actions, hence the inversion of action directions, and decide, based on location, clocks, and state variables, whether an action is allowed, in which case observation continues, or forbidden, in which case a transition to the Err state is taken. Should the observed automaton fail to act in sufficient time, thus violating a location invariant of the original specification, a τ-transition to Err will become enabled. The Err state will not be reachable if the timed traces of the observed automaton are a subset of those of the specification automaton, since each action of the former will synchronize with a valid action of the testing automaton created from the latter. Reachability analysis, in Uppaal, can determine whether Err is reachable and hence whether two timed automata are related by timed trace inclusion. A formal description of the testing construction is given by Stoelinga [2002, §A.1.5]. A tool that performs the transformation automatically for a class of Uppaal models is presented by Bourke and Sowmya [2008].

The result of applying the testing transformation to the timing diagram model is shown at the top of Figure 6 (in addition, all actions have been prefixed with a 'T' and all output transitions have been removed; the rationale and justification follow below). The original structure persists in the nodes and transitions with black lines, although

[6]Technically, the procedure verifies a timed simulation relation, but timed trace inclusion can then be inferred because it is a coarser relation [Lynch and Vaandrager 1996].

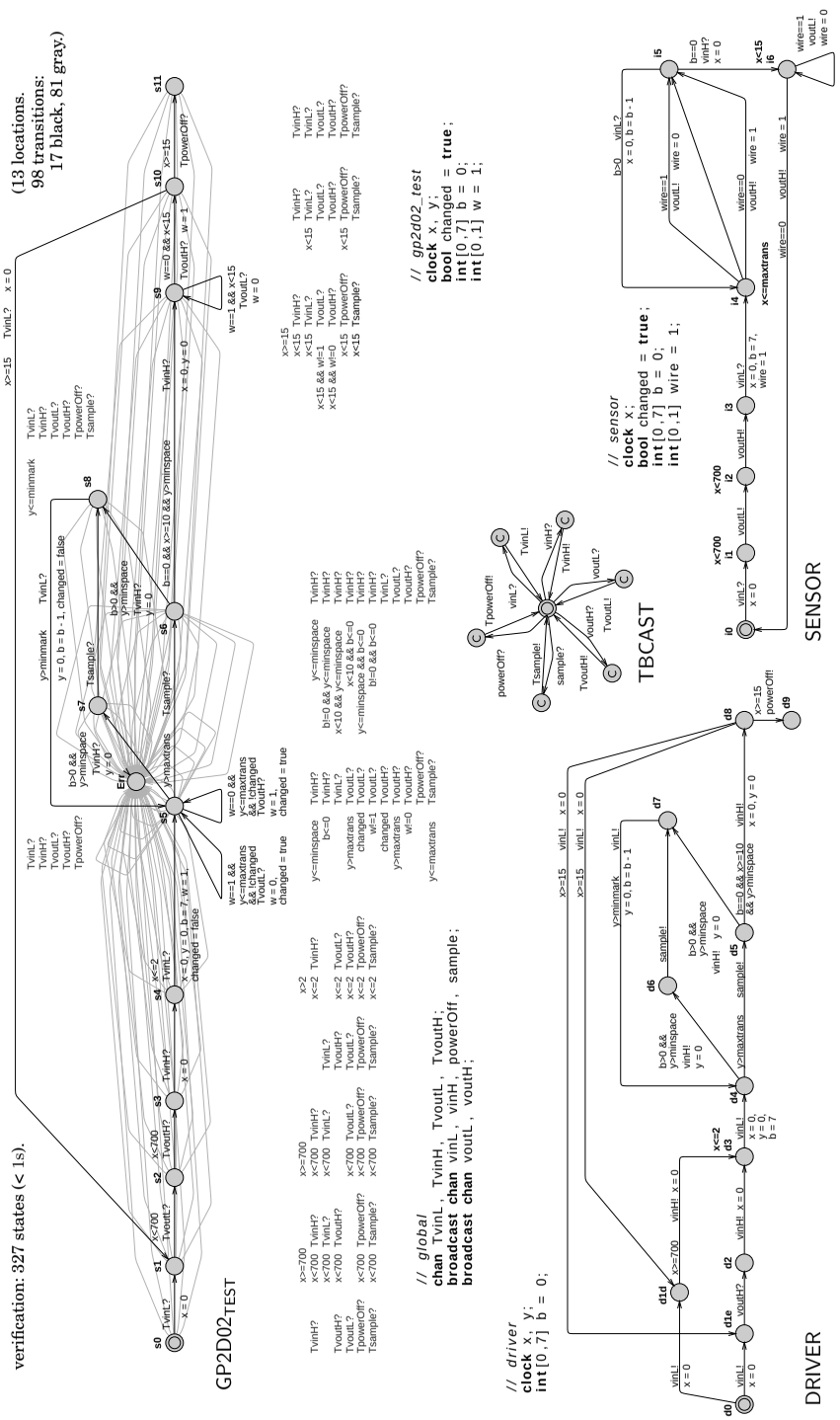
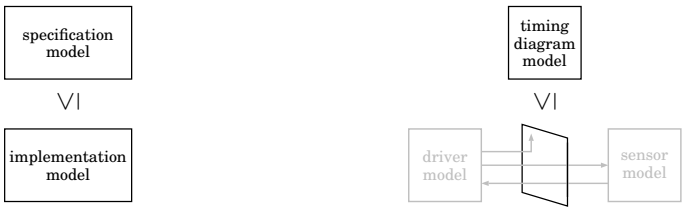


Fig. 6: Timed trace inclusion testing of the split model



(a) Normal scenario (b) The split model

Fig. 7: Deviation from the usual verification of timed trace inclusion.

the actions are now inputs rather than outputs. The new state, Err, is the hub of a nest of new gray transitions from each of the other locations. While individual transitions are not legible, the general structure is clear. Guards, where present, and actions on the error transitions are tabulated above and to the left, respectively right, of source locations s_7 and s_8 and below the others. For instance, the only action permitted from s_0 in the original automaton is vinL!, so in the testing automaton vinL? leads to s_1, and all other actions lead to Err; none of them have guards. The situation is similar from s_1, but since there is a location invariant $x < 700$ in the diagram model, a τ-transition with guard $x \geq 700$, shown as an expression with no associated action, is added to the testing automaton, and the guards of the other transitions are augmented with the invariant to ensure determinism. The guards on the error transitions are more complicated at s_5. For instance, vinH? is legal only when $b > 0$, similarly, sample? is forbidden when $y \leq$ maxtrans.

The standard testing technique is intended for comparing one timed automaton to another, see Figure 7a, but in the present case the implementation model is a network of timed automata whose interactions are to be verified, see Figure 7b. The two approaches differ from each other in their interpretation of a model as a set of timed traces. The differences have ramifications for verifying timed trace inclusion.

In the normal testing scenario, each automaton, be it specification or implementation, is given an *open* interpretation where any input or output action may occur at any time. They are assigned maximal sets of timed traces that subsume their behaviors in any composition. A normal testing automaton is always ready to synchronize on any input or output action, it effectively explores all possible behaviors of its environment.

In the split model, by contrast, there are two interacting automata, DRIVER and SENSOR. An advantage of using broadcast channels is that the output actions performed by either of the components are still observable externally even if they also synchronize with an input action of the other component (whereas successful handshake communications would become τ-transitions). Any unsynchronized input actions remain possible, but they are not supposed to be triggered from outside since they are explicitly controlled by one or other of the components.

A testing automaton must normally include both input and output actions. This is problematic for broadcast outputs because their occurrence says nothing about the behavior of the component being tested. Fortunately, they can simply be dropped from the testing automaton in Figure 6 since the split model effectively only performs output actions. Broadcast inputs would work perfectly, since they observe interactions in the split model without influencing them, but for a slight technicality: Uppaal does not currently allow clocks in guards on transitions that synchronize with input actions on broadcast channels (so called 'broadcast receivers'). For this reason only, the testing automaton must synchronize on handshake input actions, hence the 'T' prefixes, that are triggered by the new automaton TBCAST that unconditionally accepts the original

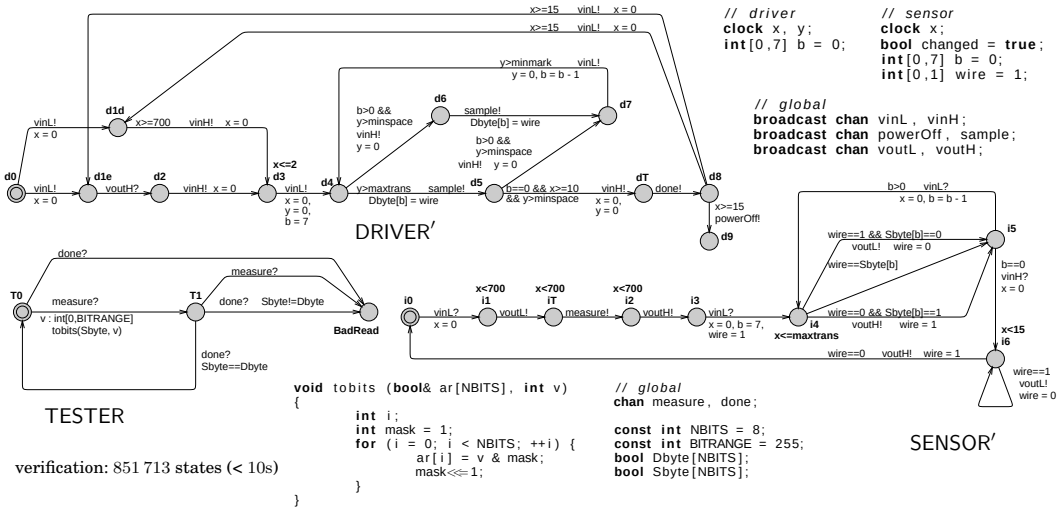


Fig. 8: Models for validating transmission correctness.

broadcast outputs and 'forwards' them to their handshake counterparts. The committed locations in TBCAST ensure that an action like vinL! is immediately followed by its counterpart TvinL!, and thus that this modification is sound.[7]

The Uppaal model for verifying timed trace inclusion of the split model against the timing diagram model is shown in Figure 6. It is a composition:

$$\mathsf{DRIVER} \parallel \mathsf{TBCAST} \parallel \mathsf{SENSOR} \parallel \mathsf{GP2D02}_{\mathsf{TEST}}.$$

Reachability analysis in Uppaal verifies that $A\Box\,(\neg \mathsf{GP2D02}_{\mathsf{TEST}}.\mathsf{Err})$, that is, the error location is not reachable. This indicates that the split model of Figure 5 correctly implements the timing diagram model of Figure 3.

4.3. Verifying transmission correctness

The driver and sensor components of the split model interact to request and perform range-readings, and then to transfer the resulting value bit-by-bit through repeated signaling and sampling over the *vin* and *vout* wires. In the previous section, it was established that the split model implements the timing diagram model. In this section, the correctness of data transmission is verified by augmenting the split model with extra details, which do not affect the protocol, expressing the transmission property as a separate automaton, and, finally, performing reachability analysis in Uppaal.

The model for verifying transmission correctness is shown in Figure 8:

$$\mathsf{DRIVER}' \parallel \mathsf{SENSOR}' \parallel \mathsf{TESTER}.$$

An array Sbyte[8] is added at the sensor component to store the bits from the most recent range-reading. The choice at i_4 between changing the *vout* level or leaving it stable is no longer non-deterministic, but depends on the bit value to be sent, that is on Sbyte[b] where b is successively decremented by the transmission loop. An array Dbyte[8] is added at the driver component to store the bits sampled from the *vout* line. The value on the shared wire variable, previously used only to restrict level changes in the sensor component, is copied into Dbyte[b] when a sample action occurs.[8]

[7]Note that no two committed locations can ever be active simultaneously.
[8]Note that the b variables are local to the components and hence distinct.

The value of Sbyte must be set when a reading occurs and compared with that of Dbyte when a transmission is complete. The TESTER automaton performs these tasks. But, importantly, the sensor itself decides when a reading has been made and, likewise, the driver decides when data has been received. These judgments are part of the interaction. New transitions are thus added to both component models. In the sensor model, a transition labeled measure! is added between locations i_T and i_2. The new location i_T has an invariant to preserve the timing properties of the protocol. In the driver model, a transition labeled with done! is added between d_5 and d_8. The altered model has a different timed trace set to the original, but the protocol remains fundamentally unchanged: dropping measure! and done! actions from every timed trace recovers the original set. The actions could also have been renamed into τs to give a set that would be equivalent to the original modulo τ-transitions.

The TESTER component synchronizes on the measure and done events. It assigns a value to Sbyte when the former occurs and compares it with the value in Dbyte when the latter occurs, entering BadRead if there is a discrepancy. A measure or done action at the wrong time also causes TESTER to enter BadRead. All possible assignments to $SByte$ are tested using a selection binding to non-deterministically choose a value in the range $[0, 2^8)$, before the tobits function converts it into eight separate bits. The same idea could be modeled using eight separate one-bit selection bindings, which would obviate the need for a conversion function but also make parametrization, over the number of bits to transmit, impossible. Reachability analysis in Uppaal verifies that $A\Box\,(\neg \mathsf{TESTER.BadRead})$ and hence transmission correctness.

Range readings are correctly transmitted in the split model. Unfortunately, this fact cannot be inferred of the timing diagram model because timed trace inclusion has only been verified in one direction. The ability to make such inferences would be another advantage of showing timed trace equivalence.

5. ASSEMBLY LANGUAGE IMPLEMENTATION

The timing diagram typically serves as a specification for writing drivers to integrate the sensor's functionality into larger designs. In §5.1, an implementation in assembly language is described and then, in §5.2, it is modeled using timed automata. In §5.3, the model is shown to be an implementation of the driver component of the split model. The timing diagram, split, and program models can thus be related by two timed trace inclusions: DRIVER $\|$ SENSOR $\leq$ TIMEDIAG and MCS51 $\leq$ DRIVER. The limitations of the assembly program model and the verification are discussed in §5.4.

5.1. MCS51 Program

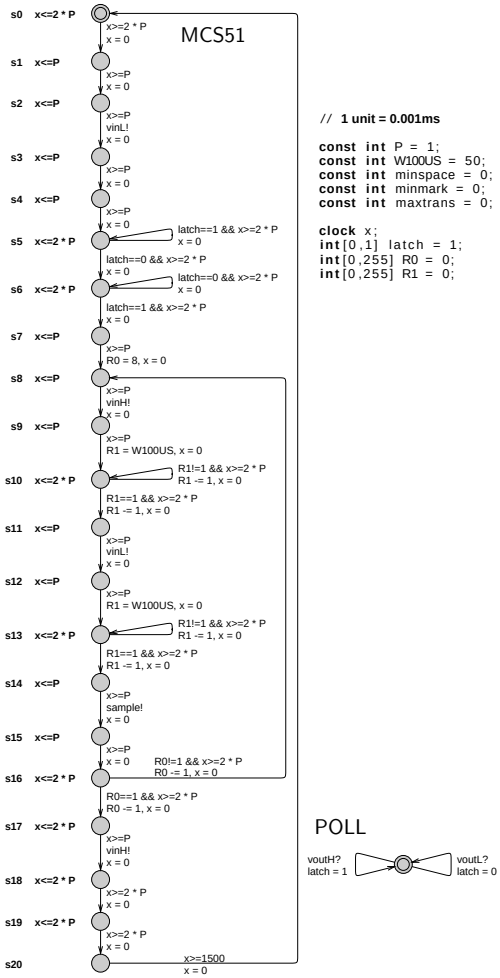
A driver for the infrared sensor, implemented in MCS51 [Intel Corporation 1994] assembly language, is presented in Figure 9a. It was adapted from a version written for a 68HC12 microcontroller [Griebling 1999]. The MCS51 architecture (8051) is older technology, but it is relatively simple, well-understood, and typical for the application domain of low-level embedded controllers. Crucially, its instruction timing is simple and predictable.[9] Instruction cycle counts can be added together without having to consider pipelining, cache effects, and other similar mechanisms. While this characteristic simplifies the timed automaton model of the next section, it also limits generalization. In any case, the aim is to examine how timed behavior is realized by the program, not to propose a general methodology for verifying assembly language programs.

The assembly program is called as a subroutine which interacts with the sensor and returns a range reading in the accumulator register. It functions as follows.

[9]Several manufacturers provide microcontrollers that faithfully preserve both the register behavior and the cycles per instruction of the 8051 instruction set; even though the period of a cycle may vary.

```
1   VIN EQU P1.0
2   VOUT EQU P1.1
3   W100US EQU 50
4
5   RREAD:  PUSH IE
6           CLR EA
7           CLR VIN
8           NOP
9           NOP
10          JB VOUT, *
11          JNB VOUT, *
12          MOV R0, #8
13  LOOP:   SETB VIN
14          MOV R1, #W100US
15          DJNZ R1, *
16          CLR VIN
17          MOV R1, #W100US
18          DJNZ R1, *
19          MOV VOUT, C
20          RLC A
21          DJNZ R0, LOOP
22          SETB VIN
23          POP IE
24          RET
```

(a) MCS51 program



(b) Timed automata model

Fig. 9: Assembly language driver implementation.

Lines 1–3. These are assembler directives that declare constants. The first two associate the vin and $vout$ signals with the two lowest pins of I/O port 0. The third defines a constant `W100US` for the number of iterations required to delay for 100 microseconds later in the program. The value depends on the number of cycles taken by certain decrement and jump instructions and the clock frequency of the target platform.

Lines 5–6 and line 23. The code is intended to execute without interruption on the target device, so it begins by pushing the Interrupt Enable (IE) register onto the stack and then disabling all interrupts by clearing the Enable All (EA) bit flag. Interrupt handling is restored, prior to returning from the subroutine, by popping the original value of the IE register which contains the EA bit. Disabling interrupts greatly simplifies modeling and reasoning about program timing, but, again, limits generalization.

Lines 7–9. Line 7 sets the vin signal to low. The following two `NOP`, *no operation*, instructions cause a brief delay before the status of $vout$ is checked. They have no

other effect. Pausing in this way between Input/Output (IO) actions is characteristic of assembly language programming and significant with respect to timing behavior.

Lines 10–11. The instruction at line 10 either jumps back to itself if $vout$ has a high value—the assembler replaces the asterisk with a relative offset—or otherwise continues to the next instruction. It continuously polls the signal state until the desired value is observed. The next line is similar, it waits for a rising transition on $vout$ by jumping back to itself while the signal value is low. The two instructions together await the occurrence of two events in sequence: voutL–voutH.

Lines 12–13 and line 21. Lines 12–21 implement a loop for receiving a reading bit-by-bit. The R0 register is the counter, it is initialized to 8 at line 12. Line 13 is labeled LOOP, so that the instruction at line 21 can jump back to it—the DJNZ, *Decrement and Jump if Not Zero*, instruction decrements the given register by one, then jumps to another location if the new value is not zero, and otherwise continues to the next instruction. The instruction on line 13 sets vin high, causing a rising transition: vinH.

Lines 14–15. Between the rising transition on vin and a subsequent falling transition there is a delay of approximately 100 microseconds, which is implemented by counting down from the W100US value in the R1 register. The delay is equal to the amount of time it takes to execute the DJNZ instruction multiplied by the initial value of the counter register. Such loops, with no internal statements, serve to delay program execution.

Lines 16–18. These lines set vin to low, giving a vinL action, and then delay for another 100 microseconds.

Lines 19–20. Together, these lines sample the level of $vout$ into the received range reading. The former copies the bit named VOUT into the carry flag. The latter shifts the contents of the accumulator to the left and sets its LSB to the carry flag value.

Lines 22–24. Finally the driver sets vin high, restores the interrupt enable register, and returns to the calling program.

5.2. Program model

The assembly language program is modeled as the composition of two timed automata: MCS51 ‖ POLL. The model is presented in Figure 9b.

Since only integer timing constants are allowed in Uppaal, the timing diagram and split model assumed a scale of one unit of model time to 0.1ms of real time. It turns out that even older microcontrollers, like those of the MCS51 family, are much faster and thus require a smaller time scale: one unit of model time in the program model is equivalent to 0.001ms of real time. The number was chosen for a device clocked at 12MHz with a machine cycle every 12 clock periods and running at one cycle per microsecond [Intel Corporation 1994, p. 1-18]. For the driver model to serve as a specification, all of its constants must be multiplied by 100.

The program model is based on a direct translation from the source program. The translation effectively gives a semantics for programs written in the MCS51 instruction set in terms of timed automata. Close structural similarity between the source program and its model is important because results obtained for the latter are inferred of the former. In other words, that the model is a faithful formalization of the program is determined solely by informal argument: the larger the gap between them the greater the risk of an error. A location s_n in the model of Figure 9b corresponds to the instruction at line $n - 5$ in the program of Figure 9a.

It takes a fixed number of cycles to execute each MCS51 instruction. These delays and the propulsion of execution are expressed in the model by transition guards and location invariants. The constant P is the length of a single execution cycle. It varies across MCS51 implementations and oscillator frequencies. The mapping, though, from instructions to the number of execution cycles is fixed for the instruction set [Intel Corporation 1994, Table 10]—the informal semantics thus describe not only how in-

structions transform microcontroller states but also how long those transformations take, at least nominally. Lower and upper bounds on instruction execution times are expressed as inequalities between the clock x, which is reset on every transition, and integer multiples of P. For instance, the ⟨**PUSH** IE⟩ instruction takes two cycles to execute, thus the invariant at location s_0 is $x \leq 2{*}P$, which forces progress, and the guard on the outgoing transition is $x \geq 2{*}P$, which expresses the execution time.

5.2.1. Modeling time. According to the translation, instructions begin and end at precise multiples of P. While this simplification is adequate for present purposes, since the timing constants in the program model are an order of magnitude smaller than those in the timing diagram model, it ignores two potentially important aspects: oscillator inaccuracies and processor states within machine cycles.

No oscillator is perfect. At the very least such 'clocks' drift with respect to one another and against an ideal notion of real time. Such inaccuracies can be important in certain applications, and, more fundamentally, their existence makes the decision to conflate processor time and real time in the program model doubtful. They could be incorporated into the program model by increasing the resolution of model time and widening invariants, for instance to $x \leq P + \epsilon$, and guards, $x \geq P - \epsilon$. In this way, the (still idealized) behavior of the execution platform would be encoded in the program semantics, and any approximations, even just $\epsilon = 0$, would be clearly stated. Rather than blend the different aspects of platform and program so implicitly, execution platforms could be modeled separately and composed with program models; the models are then not only individually reusable, but each is also likely more comprehensible, the intricacies of their interrelationships being left to the mechanics of the modeling language. In one approach [Vaandrager and de Groot 2006], the cycle clock is modeled as a separate automaton that emits a tick action at intervals. The disadvantage, compared to simply widening the invariants, is that synchronizing with tick on each instruction transition makes it awkward in Uppaal to also synchronize with other actions, as occurs for inputs and outputs in the program model.

Aside from oscillator inaccuracies, the program model also abstracts from the detailed timing behavior of an MCS51 device [Intel Corporation 1994, pp. 1-17–1-20]. Each oscillator period corresponds to a *phase*, two phases make a *state*, and six states make a *machine cycle*. Instructions are fetched and executed at specific phases within a cycle. Significantly, values are written or sampled from ports in specific phases [Intel Corporation 1994, pp. 3-33–3-35]. While many applications depend on the time taken to execute individual instructions, it is doubtful whether the correctness of a system should further depend on the finer timing details within a cycle. But such judgments are perhaps best made by engineers for each specific application.

5.2.2. Features of the model. The program model contains several other interesting features: input events, output events, loops for delaying, and assumptions on how frequently range-readings are requested.

The program treats $vout$ as an input line: Sometimes it reacts to changes in the signal level. Sometimes it samples the signal level. It would not be accurate to label transitions that represent instructions with voutL? or voutH? actions because their occurrence is restricted to particular instants of time, whereas not only may events of both types occur at any time, but, for the model to be input-enabled, they must be allowed to occur at any time. Instead, a separate automaton POLL is introduced. It effectively models the hardware latches of the IO port connected to $vout$. It is always ready to synchronize with voutL?, setting a variable latch to zero, and voutH!, setting latch to one. The assembler instructions that poll $vout$, for either a high level ⟨**JB** VOUT, $*$⟩ or low level ⟨**JNB** VOUT, $*$⟩, are modeled as loops, at s_5 and s_6 respectively, that poll the latch variable within the timing constraints of instruction execution. The assembly

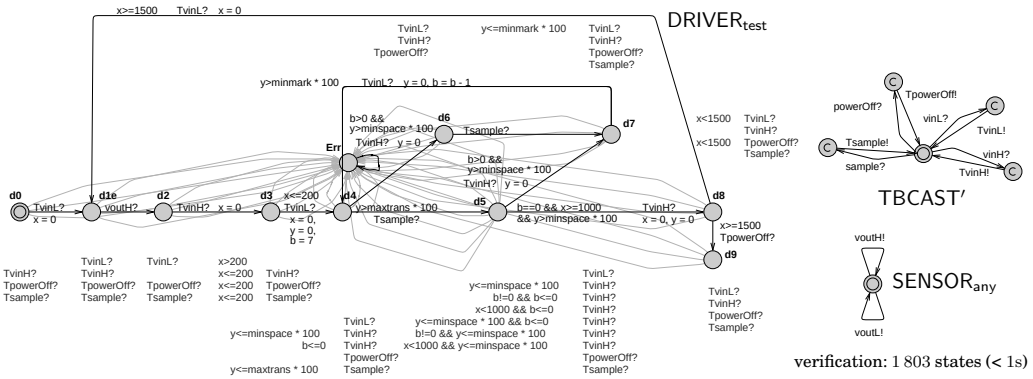


Fig. 10: Timed trace inclusion tester for $\mathsf{DRIVER}_{\mathsf{evt}}$.

program waits for a rising transition on $vout$ by first blocking until the latched value is zero and then polling again until it is one.

Output actions, on the other hand, are justifiably constrained by the program model. Thus, and due to the assumption that outputs are never refused, the transitions for certain instructions can simply be labeled with output actions: ⟨**SETB** VIN⟩ becomes vinH!, ⟨**CLR** VIN⟩ becomes vinL!, and ⟨**MOV** VOUT, C⟩ becomes sample!.

The loops at s_9/s_{10} and s_{12}/s_{13} cause the program to pause between changes to vin. The register R1 is first initialized and then decremented until it reaches zero, which accumulates individual $2{*}$P delays giving a total delay of W100US$*2*$P. The loops do not change the microcontroller state in any significant way, but yet they cannot be removed without changing the observable behavior of the model. This is in contrast to programming language semantics where such instructions could be shown equivalent to an instruction that does nothing, for instance to **skip** or ⟨**NOP**⟩.

The guard on the last transition from s_{20} back to s_0, $x \geq 1500$, expresses a required minimum delay between calls to the subroutine. As there is no location invariant on s_{20}, the model allows any finite non-zero number, or even an infinite number, of repetitions.

5.3. Verifying the program model

Not only does modeling the program offer insight into the assembly program, execution platform, and instruction set semantics, but the model itself can be validated against the timing diagram and checked for other desired properties.

The driver component of the split model is modified to serve as a specification for the program model: all of the constants are multiplied by 100, to account for the increased resolution of model time, and the top path through d_{1d} is removed to make the model deterministic. Determinism is required by the timed trace inclusion testing construction. Since the assembler program does not wait for 70ms or more after requesting a range reading but rather begins clocking data in response to an initial voutL, the unused possibility is simply removed from the model. This is sound for timed trace inclusion because it reduces the set of timed traces of the specification.

A testing automaton for the driver specification is derived by slightly extending the approach of §4.2.2: an automaton is built using the standard technique, all output transitions to the Err location are removed, and all input actions are prefixed with 'T', see Figure 10. This automaton must run in parallel with TBCAST′ to work around the Uppaal limitation on clock guards in broadcast receivers. Additionally, since the driver is open to the inputs voutL? and voutH?, the tester must include corresponding outputs. This explains the addition of $\mathsf{SENSOR}_{\mathsf{any}}$, which effectively adds self-loops for

the two actions to all states of the tester except d_{1e} which has a self-loop on voutL!, and a transition elsewhere on voutH!. This modification is justified because the driver must be input-enabled for these outputs. It only works because, in this case, there is no clock guard on the transition between d_{1e} and d_2; were this not so a TvoutH! action and explicit self-loops would be required.

The relation between the driver specification and program models can be verified by model checking the property: $A\Box\,(\neg \mathsf{DRIVER}_{\mathsf{test}}.\mathsf{Err})$ on:

$$\mathsf{MCS51} \parallel \mathsf{POLL} \parallel \mathsf{DRIVER}_{\mathsf{test}} \parallel \mathsf{TBCAST}' \parallel \mathsf{SENSOR}_{\mathsf{any}}.$$

Uppaal quickly shows that the property holds, and thus that the timed traces of the program model are included in those of the driver model. But what about the original timing diagram? Importantly, it is the composition of driver and sensor models that was verified against the timing diagram: there is a possibility that the driver model has behaviors which violate the timing diagram model, but which do not occur in composition with the sensor model. The verification thus shows only that the program model is correct against the timing diagram model, and by inference the assembly program against the specification, provided that it is used with a correct sensor, namely one whose timed traces are a subset of those of the sensor component of the split model.

Timed trace inclusion gives no guarantees about liveness, and such properties must be checked separately. Both deadlock freedom, $A\Box\,(\neg \mathsf{deadlock})$, and that the routine has the possibility of running to completion, $E\Diamond\,(\mathsf{MCS51}.s_{20})$, hold when the program model is placed in parallel with the sensor component with scaled constants:

$$\mathsf{MCS51} \parallel \mathsf{POLL} \parallel \mathsf{DRIVER}_{\mathsf{evt}} \parallel \mathsf{SENSOR}_{\times 100}.$$

The property that the routine always completes once called, $\mathsf{MCS51}.s_0 \rightsquigarrow \mathsf{MCS51}.s_{20}$, does not hold: from i_1 the sensor can perform a voutL! and voutH! before the program starts polling at s_5, which does not violate DRIVER because nothing forces a vinH! from d_2. Lower bounds on the sensor actions are needed.

5.4. Limitations

The choice of such a relatively simple execution platform restricts the number of issues that must be addressed and simplifies both the approach and exposition. But it also means sidestepping the most difficult aspects of timing on modern processors, where execution paths and cache contents must be considered. Moreover, the model itself does not capture all the timing intricacies of the MCS51, and the question of when such complexities can be ignored completely, or distilled to more abstract principles, and whether they must ever be modeled in complete detail merits further consideration.

Another limitation of the approach is that it ignores interrupts and the concomitant timing complexity. For instance, delays in the program are implemented by busy waiting rather than with timers. Furthermore, the possibility of being interrupted by other events or concurrent processes is not considered. While interrupts are avoided in some approaches to embedded design, they are widely used in practice as a lightweight means of scheduling, and as a way to reduce latencies and improve timing accuracy.

The need for so much timing accuracy is indeed questionable, since the program turns out to be two orders of magnitude faster than the timing constraints in the timing diagram. The challenge is to create delays rather than to meet deadlines, which underscores the importance of programming with time but neglects an important aspect of programming embedded systems.

Lastly, just as the sensor is usually just a single component of a system, so is its driver just a single routine in a program. The issue of composing different routines, each with their own timing behaviors and constraints deserves more attention.

6. DISCUSSION

The case study and its limitations. The detailed case study presented in this article is a concrete example of an application of rigorous modeling and analysis methods to a realistic, if small-scale, embedded component. It exposes the peculiarities of one specific problem, but also offers more general insights into the application domain. While the example suffers several limitations, the limitations are interesting in themselves and could help to guide and evaluate future case studies. It is challenging, however, to extract general insights from the study of arbitrary examples; some peculiarities may be truly idiosyncratic to a particular example. On the other hand, the design and implementation of embedded systems are characterized by unconventional devices and fine detail—research that ignores too many asperities risks irrelevance.

While no pretense is made of proposing a general methodology, this article does contribute a specific example to the growing collection of applied real-time verifications, and it seems reasonable to make five general observations. First, time is integral to some behavioral specifications and not simply a nonfunctional requirement for later design stages. Second, while timing constraints can rule out sequential behaviors, as exemplified in the correctness argument for Fischer's Protocol [Abadi and Lamport 1994, §3.4], this is not their only purpose. Third, timing behavior is not just about meeting deadlines; determining precisely when an action occurs is also important and some commands are only given for their effect on timing. Fourth, timing behavior arises from an interaction of programming language semantics and platform characteristics; creating models from the latter involves choosing abstractions that are suitable for the application at hand. Last, even a simple timing diagram can express quite complex relations between events and present difficulties of interpretation.

The specification models. It would be difficult to argue that the model of Figure 3 is easier to read than the timing diagram on which it is based, but it is certainly more precise. The advantages of informality in timing diagrams must be balanced against the need for accuracy during implementation and analysis. While timing diagram languages are not considered in this article, it is clear that the modeling process involves more than simple transcription: careful analysis, interpretation, and the occasional assumption may be necessary. Indeed, rather than clutter timing diagrams with the additional constraints needed for precise formalization, it may be better to start with a detailed formalism, like timed automata, and then to generate, according to manual instructions, sets of timing diagram 'traces' to serve as documentation.

While timed trace inclusion of the split model in the timing diagram model was verified, the inverse relation was not. Using an approach based on timed games and a new model-checking algorithm, Bourke et al. [2011] present an altered split model and show full timed trace equivalence between it and the timing diagram model. The use of explicit testing automata nevertheless remains a simple and traceable alternative.

Real-time programming. The assembly language modeling and verification exhibited in this article is unlikely to scale. Verifying assembly language programs of realistic size requires specific abstractions and techniques even when real-time behavior is ignored [Schlich 2008]. Furthermore, the approach does not address interrupts, the inclusion of driver routines in larger systems, or sophisticated execution platforms.

It may be interesting to try to program the driver using different approaches, with two requirements: that programs are verified against the timing diagram model, and that programs can be faithfully executed against a real sensor. The adjective 'faithfully' implies an uncompromising relation between the verified object and a running system! Proposals should be evaluated on the compromise made between adequately precise timing behavior and the interrelated issues of abstraction and portability. 'Abstraction' essentially means comparing the text of a program with the behavior it seeks

to express, and 'portability' relates to the ability to reuse the expressed solution whilst maintaining the timing behavior, at least within the constraints of the original specification. The proposed assembly program can be executed and verified, and it has precise timing behavior but it is neither very abstract, since the desired behavior is really only expressed as a side-effect of a nest of implementation-specific detail, nor is it especially portable, since it specifies a very rigid trace of the timing diagram—the flexibility allowed by the problem constraints cannot be further exploited.

ACKNOWLEDGMENTS

Several iterations of the model of §3 were presented at 'timed automata teas' at NICTA during which Ansgar Fehnker, Peter Gammie, and Rob van Glabbeek asked insightful questions and made helpful suggestions.

The rest of the work in this paper was shaped during a pleasant afternoon at Radboud University in 2006 where Frits Vaandrager proposed both the approach used in §4.3 to test transmission correctness, and the timed trace inclusion testing technique applied in §§4.2 and 5.3.

We thank the reviewers for their valuable recommendations and comments.

REFERENCES

ABADI, M. AND LAMPORT, L. 1994. An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems (TOPLAS) 16,* 5, 1543–1571.

ALUR, R. AND DILL, D. L. 1994. A theory of timed automata. *Theoretical Computer Science 126,* 2, 183–235.

BOURKE, T. 2009. Modelling and programming embedded controllers with timed automata and synchronous languages. Ph.D. thesis, University of New South Wales, Sydney, Australia.

BOURKE, T., DAVID, A., LARSEN, K. G., LEGAY, A., LIME, D., NYMAN, U., AND WĄSOWSKI, A. 2011. New results on timed specifications. In *Recent Trends in Algebraic Development Techniques, 20th Int. Workshop, WADT 2010 (Revised Selected Papers)*, T. Mossakowski and H.-J. Kreowski, Eds. Lecture Notes in Computer Science Series, vol. 7137. Springer-Verlag, Schloss Etelsen, Germany, 175–192.

BOURKE, T. AND SOWMYA, A. 2008. Automatically transforming and relating Uppaal models of embedded systems. In *Proceedings of the 8th ACM Int. Conference on Embedded Software (EMSOFT'08)*. ACM Press, Atlanta, Georgia USA, 59–68.

GRIEBLING, E. T. 1999. GP2D02 assembly language driver for 68HC12B32 microcontroller. `http://home.earthlink.net/~tdickens/68hc11/code/sharpirhc12.asm`.

HENZINGER, T. A., NICOLLIN, X., SIFAKIS, J., AND YOVINE, S. 1994. Symbolic model checking for real-time systems. *Information and Computation 111,* 2, 192–244.

INTEL CORPORATION. 1994. MCS®51 microcontroller family user's manual.

JENSEN, H. E., LARSEN, K. G., AND SKOU, A. 2000. Scaling up Uppaal: Automatic verification of real-time systems using compositionality and abstraction. In *Proceedings of the 6th Int. Symposium on Formal Techniques for Real-Time and Fault-Tolerance (FTRTFT '00)*, M. Joseph, Ed. Lecture Notes in Computer Science Series, vol. 1926. Springer-Verlag, Pune, India, 19–30.

KAYNAR, D. K., LYNCH, N., SEGALA, R., AND VAANDRAGER, F. 2006. *The Theory of Timed I/O Automata*. Synthesis Lectures on Computer Science. Morgan & Claypool Publishers, San Rafael, CA, USA.

LARSEN, K. G., PETTERSSON, P., AND WANG, Y. 1997. Uppaal in a nutshell. *Int. Journal of Software Tools for Technology Transfer 1,* 1–2, 134–152.

LYNCH, N. AND VAANDRAGER, F. 1996. Forward and backward simulations. Part II: Timing-based systems. *Information and Computation 128,* 1, 1–25.

RAMSEY, A. 2001. Interfacing the GP2D02 to a Microchip PIC. Encoder: The Newsletter of the Seattle Robotics Society.

SCHLICH, B. 2008. Model checking of software for microcontrollers. Ph.D. thesis, RWTH Aachen University, Aachen, Germany.

SHARP CORPORATION. 1997. GP2D02: Compact, high sensitive distance measuring sensor.

STOELINGA, M. I. 2002. Alea jacta est: Verification of probabilistic, real-time and parametric systems. Ph.D. thesis, Katholieke Universiteit Nijmegen, The Netherlands.

VAANDRAGER, F. AND DE GROOT, A. 2006. Analysis of a biphase mark protocol with Uppaal and PVS. *Formal Aspects of Computing 18,* 4, 433–458.

Received May 2011; revised December 2011; accepted April 2012