A Compositional Analysis of Broadcasting Embedded Systems

Michael John Brockway

A thesis submitted in partial fulfilment of the requirements of the University of Northumbria at Newcastle for the degree of Doctor of Philosophy

June - September 2010

Abstract

This work takes as its starting point D Kendall's CANdle/bCANdle algebraic framework for formal modelling and specification of broadcasting embedded systems based on CAN networks. Checking real-time properties of such systems is beset by problems of state-space explosion and so a scheme is given for recasting systems specified in Kendall's framework as parallel compositions of timed automata; a CAN network channel is modelled as an automaton. This recasting is shown to be bi-similar to the original bCANdle model.

In the recast framework,"compositionality" theorems allow one to infer that a model of a system is simulated by some abstraction of the model, and hence that properties of the model expressible in ACTL can be inferred from analogous properties of the abstraction.

These theorems are reminiscent of "assume-guarantee" reasoning allowing one to build simulations component-wise although, unfortunately, components participating in a "broadcast" are required to be abstracted "atomically". Case studies are presented to show how this can be used in practice, and how systems which take impossibly long to model-check can tackled by compositional methods.

The work is of broader interest also, as the models are built as UPPAAL systems and the compositionality theorems apply to any UPPAAL system in which the components do not share local variables. The method could for instance extend to systems using some network other than CAN, provided it can be modelled by timed automata. Possibilities also exist for building it into an automated tool, complementing other methods such as counter-example-guided abstraction refinement.

Contents

Acknowledgements

I should like to acknowledge the support of my colleagues and supervisors David Kendall (on whose work I have built), William Henderson and Adrian Robson for their unstinting support throughout the period of my work on this thesis. Not only have they given much valuable and wise advice, but they have been there to challenge me to explain myself – so many things have become clear to me after that.

Thanks also to the School of Computing, Engineering and Information Sciences (and formerly, the School of Informatics) of Northumbria University for meeting my costs, and to all my colleagues in the School for all their encouragement and advice.

I am grateful, last but not least, to my family, especially Rosie, Rob, Josy and Barbara for their loving support and encouragement during the many times my work has spilled over into their lives, and for their constant reminder that there is life outside academia.

Declaration

I declare that the work contained in this thesis has not been submitted for any other award and that it is all my own work.

Chapter 1

Introduction and Overview

1.1 Embedded Systems

Embedded Systems are electronically controlled appliances which include a computer – usually termed a *microcontroller* in the electronic engineering community – or several such computing nodes in communication over one or more *networks*. In the latter case one may speak of a *distributed* embedded system or, when (as is often the case) the communication is broadcast rather than point-to-point, a *broadcasting* embedded system. Such appliances abound: examples include toys, electronic games, home appliances, Hi-Fi, TV and DVD equipment, computer peripherals and mobile phones. Cars employ a network of microcontrollers for engine management, anti-lock breaking and the like, and distributed embedded systems are common in industry (controlling production lines for instance) and in medicine, science and high technology. Statistically almost all (97-99%) manufacture of computer CPUs is for embedded systems rather than for "traditional" computing systems.

These systems do not *look like* computing equipment to the casual observer. The computers are in the background. They boot when the appliance is switched on and their software is expected to run reliably, "seamlessly" until it is switched off. Their software generally includes a number of concurrent *duty cycles*, each a repeating loop in which inputs are obtained from elec-

tronic *sensors*, computation is performed, and outputs delivered to electronic *actuators*.

The outputs are generally required to meet real-time *deadlines* and often, also to perform at high speed. A multimedia system, for example, must process megabytes of data every second. Failure of the multimedia system to meet a real-time deadline may not be a disaster, however: it may just mean a degradation in output quality. This is an example of a *soft* real-time system. A *hard* real-time system is one in which failure to meet a deadline implies a total failure of the system. This may be a mission-critical failure in the case of an industrial robot, a communications satellite or a rover deployed on Mars; or worse, a safety-critical failure, as in the case of nuclear reactor control system, a medial radio-therapy machine, or and anti-lock breaking system of a car.

Such a failure may not be a simple consequence of too-slow sensing, communication or computing other such inadequate provision of hardware resources, and also may not be a straightforward consequence of algorithmic complexity. If we assume the hardware is adequate to the task, *concurrent* software might fail because of unforseen *interference* between the concurrent tasks. A flaw in the logical design could in certain circumstances lead to *unsafe* behaviour or to *deadlock* or some other failure of *liveness*. How can one be sure of having thoroughly tested for such flaws? Techniques exists for investigating the *stochastic* behaviour of these systems, for measuring the *probability* of such failure; but in a safety critical system, we would like to know whether *a failure is possible at all*, and if so, what combinations of circumstances lead to failure.

1.2 Model Checking

Formal methods have a role to play in this. Constructs of a mathematical character may be employed to aid the construction of precise requirements statements while analysis proceeds informally. Greater rigour than this is possible if a formal language or process algebra is employed to model the system or some part of it, and to express specifications, desired properties.

Software tools may then be employed to check syntax, data types and to simulate or animate the system. Greater rigour still is possible if the formal language has rigorously defined semantics which can be checked with software tools. Wolper [114] has argued for this level of rigour even at the cost of limited applicability.

A well established method is to model a software system as a "state machine" or *automaton* and employ model-checking software to explore the *state space* of the automaton, checking for deadlock or unsafe states or behaviours. Desirable or undesirable behaviours can be expressed as automata and their *composition* with the system model checked for their necessity, contingency, possibility or impossibility. Behaviours can be expressed as formulae in a variety of modal *temporal logics* from which the checking software tools can generate the composite model and check it. An introduction to temporal logics and model checking tools and techniques is given by [26], and a more advanced account in [45].

The work outlined in this thesis aims to show how such model-checking can be aided by *compositional* techniques. Models of real systems are generally compositions of several parallel, concurrent component processes and while the system model as a whole may be too complex to check, it is possible to infer properties of the whole from verifiable properties of its components.

Concurrent system models which do not contain explicit reference to time are straightforward to check: there are established methods for keeping the computational complexity of the state space search within reasonable bounds. More problematic are models where time plays an explicit role. These can be modelled with *timed automata* (§2.2) which feature not only control *locations* and *transitions* between locations, but also one or more *clock* variables and, at each location, an *invariant*, a predicate on the clock variables which must be true there, and, for each transition, a guard: a predicate that must be satisfied for the transition to be enabled.

A simple example is depicted in figure 1.1, in which there are two locations A, B, a clock variable h and another discrete variable x. The process begins at location A having initialised h, x both to 0. The predicates in braces are invariants: the process may reside at location A where clock $h <= 10$ time

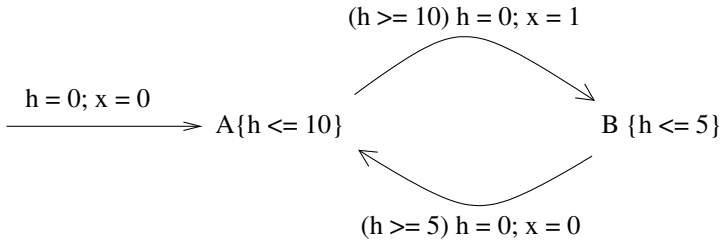


Figure 1.1: A simple timed automaton

units. The transition $A \rightarrow B$ may be taken as soon as clock h reaches a value of 10, and when it is taken, the effect is to reset h to 0 and update x to 1. Thus, the running process alternates between A, B; variable x alternates between values 0, 1 for 10, 5 time units respectively. *Non-determinacy* results if, for instance, the invariant at A is relaxed to $h <= 15$ and the $A \rightarrow B$ guard to $h >= 5$.

Many properties of timed automata are decidable, although the computational complexity of the decision may grow unacceptably fast. However, decision problems become harder with *stopwatch automata* [43] where clocks may start and stop running – these are needed, for instance, to model preemptive scheduling – and with *hybrid* automata [42, 67] which carry data variables in addition to clocks. Data variables suffer updates on discrete action transitions and vary according to some rate of change as time passes at some location. See, for instance, [67].

Software tools exist, even so, for verifying properties of such structures. We use UPPAAL [50, 89]; other tools in which I have been/am interested include CMC [87], Thao Dang's [49] tool "d/dt", CADP [58, 59] (supporting timed extensions to LOTOS [92]), and process-algebra-based tools such as the Edinburgh Concurrency Work-bench.

David Kendall [79] developed a modelling method and language, bCANdle, particularly suited to modelling broadcasting embedded control systems communicating via *controller area networks* (CAN). The particular interest in CAN derives from the development of this network technology for the automotive industry and its subsequent spread to other industrial control

applications. The method derives a *timed transition system* from a model defined in the bCANdle language and an associated tool is able to check safety and liveness properties against it. He shows that a model in bCANdle is semantically equivalent to a timed automaton.

The challenge is to scale up to "industrial"-sized models. All these tools work for small problems and models, but the computational complexity explodes as we scale up.

One way to combat the "state space explosion" problem is to model larger systems as composites of smaller systems in parallel and to examine how properties of a composite might be inferred from properties of its components – see §3.4. The work of the present thesis is to investigate such *compositional* approaches and to apply them to bCANdle.

The bCANdle framework is recast so that a model is, semantically, not a single monolithic timed automaton as originally proposed, but rather, a parallel composite of processes. The CAN network channels appear as processes running in parallel with the processes defined by the system.

This recasting is shown to be semantically equivalent to the original bCANdle, and the parallel components are devised in such a way that compositional methods, such as assume-guarantee reasoning may be employed to infer properties of modelled system from properties of the components, or of systems in which components of the original system are replaced by simpler *abstractions*.

We thus show how, in general, the behaviour of broadcasting embedded systems modelled in bCANdle may be reasoned about by inferring from behaviour of components. We show, indeed, that by using such compositional reasoning, checks can be performed using less processing time than was possible in the non-compositional approach, in some cases even succeeding where a non-compositional check fails to terminate. Broadcasting embedded systems communicating by CAN is a special case deemed by Kendall sufficiently important to merit the development of his CANdle modelling framework; but the work presented here in fact applies not just to CAN-communicating systems but, more generally, to any systems that can be modelled as parallel composites of timed automata, whether or not a network is involved (and

whether or not is is a CAN network).

As far as we are aware, this is the first time such compositional approaches have been applied to *broadcasting* embedded systems. At any rate, a compositional methodology which allows us to reason about broadcasting embedded systems seems a worthwhile addition to bCANdle.

A parallel-composite model derived in this way from a bCANdle description is readily checked by the UPPAAL model-checking tool and indeed the compositional rendering of bCANdle developed in this thesis makes sense as a composite of UPPAAL-style timed automata.

In the course of deriving this compositional recasting of bCANdle, it became apparent that one could develop UPPAAL (or timed automaton) based models of broadcasting embedded systems by developing the automata directly rather than going via bCANdle, and making use of an automaton representing a CAN channel. The examples discussed in chapter 8 were developed in this way, and indeed this turned out to be the most natural way to make the abstractions employed in so-called *assume-guarantee* reasoning. See sections 3.4.1, 3.4.2, 3.4.3 for a brief introduction to this component-wise method of reasoning about composite systems.

The compositionality theorems employed here apply to any parallel composite of timed automata. This work therefore has potentially broader application than just to CAN-based broadcasting embedded systems: to any systems modellable in UPPAAL, in fact, or in analogous formalisms to which assume-guarantee theorems apply. A suitable model of a CAN channel is provided and could, if desired, be extended or modified (or replaced altogether) to model other network or interprocess communication protocols.

Of course, for CAN-based systems, starting with bCANdle is still an option. An interesting future project is to develop a software tool, perhaps to use *counterexample-guided abstraction refinement* (CEGAR, §3.5, §9.3.1) to generate abstractions. bCANdle is then a good candidate for an input language if we are concentrating on CAN-based systems, although other ways of specifying timed automata are also possible. In the present work, a simple language was developed, based on the UPPAAL `.xta` format, in which automaton-based models of CAN-based systems can easily be specified. A

simple tool expands these specifications to full `.xta` format which can be input to the UPPAAL verifier. Future work is to devise software to integrate this approach and incorporate CEGAR.

We do not yet have such an "automatic" tool; the present work is done by hand. However, the compositional approach has been proved in principle and can evidently be applied to any concurrent real-time system modellable by timed automata.

1.3 Scope of the Thesis

The focus of the thesis is on the real-time behaviour of broadcasting embedded systems which communicate via controller area networks (CAN) – the subject matter of bCANdle – although the work could be extended to systems using other communications technologies.

Timed automata provide a model of concurrency in terms of the arbitrary interleaving of actions: concurrent actions a, b are modelled as a nondeterministic choice of a followed by b and b followed by a. Arguments are made for alternative methods, such as Petri nets or the *event structures* described by Bowman and Gomez [29] which model "true" concurrency. It is sometimes argued (see, e.g. Bowman and Gomez [29] chap 4) that modelling true concurrency gets one closer to the physical implementation of an embedded system and to the fine structure of its decomposition into components, possibly allowing the modelling of questions of the causal independence or autonomy of the concurrent actions a, b, for instance. In general, an "interleaving" view of concurrency abstracts away from the "physical" distribution or disposition of the embedded system and provides an emphasis on *observable behaviour* and a "global" view while a true-concurrent model makes possible a more "local" view of the system. One can view a model based on interleaving concurrency semantics as a "black box" view, while a model based on true concurrency affords a "white box" view. Both types of model have their strengths and are appropriate at different levels of abstraction. Thus, interleaving concurrency is most appropriate at early stages of system design where global system requirements are being identified, while true

concurrency is more applicable where when system components are being identified and implemented.

The work of this thesis adopts the former, an abstract view of the systems. We are looking at observable behaviour; we are not looking at user interfaces; nor are we necessarily concerned with every computational detail. We *are* interested in timed behaviour of models of systems in which detail has been abstracted away. We are interested in, for instance, questions such as whether a value of a variable is guaranteed to be communicated within a real-time deadline; whether a feedback or closed-loop control system is effective in keeping some quantity within safe bound; whether a specified timed property of a composite, concurrent, distributed system is *guaranteed*.

We take an abstract view of such properties also. These will be formally expressible in *temporal logic* – in particular, in CTL (see §2.1.3). Safety properties can considered by expressing them contrapositively in the form "for all system executions, at all times, φ", where φ expresses safety in terms of system variables. Liveness (livelock freedom) similarly can be expressed in the form "for all system executions, at some time, φ".

It is key to the work that such properties, and more general ones such as starvation freedom or fairness, can be expressed in ACTL, the universally quantified fragment of CTL.

We are not directly concerned with system performance in general (other than safety, liveness and the like), nor with stochastic behaviours.

The major task of the thesis is to extend Kendall's bCANdle formalism and model-checking techniques by recasting bCANdle in a compositional style and showing how compositional techniques might then be employed in reasoning about bCANdle models of CAN-broadcasting embedded systems. Although other formalisms and techniques are available (surveyed in chapter 3), we concentrate pretty much on the timed automata and timed transition systems that underpin bCANdle. As chapter 8 shows, compositional techniques such as *assume-guarantee* reasoning do afford means of checking of models for properties by software tools more quickly than would the case with a "direct" check and perhaps, feasible where a direct attack would run out of memory.

1.4 Contribution of the Thesis

Kendall's bCANdle (§3.2.2) is a very useful framework for modelling, specifying and checking CAN-based embedded real-time systems. The present work extends it by recasting it in a semantically equivalent form as a parallel product of timed automata.

This allows us to apply *compositional reasoning* techniques, such as assume-guarantee reasoning (§3.4).

The timed-automata modelling approach has been extended to work with UPPAAL "systems" of "processes" – essentially parallel products of timed automata.

Theorems have been developed allowing one to infer a *simulation* (§2.4) relationship between such a parallel product and some abstraction of it from simulations that exist between the components of the product and abstractions of them. The benefit of this is that a property of the product may then be inferred from the same property of the abstraction, provided the property is expressible in terms of universal quantification over execution traces – for instance, properties expressible in ACTL and their timed analogues. This is useful if the abstraction has a smaller state space than the original model: the check of the property on the abstraction will then be faster than the same check on the full model, and perhaps feasible on the abstraction whilst infeasible (due, say to memory demands) on the full model.

Small case studies of this are presented, showing the kinds of abstraction and simulation that are useful.

In a timed setting, many properties – safety, liveness, livelock freedom, freedom from starvation – may be cast as reachability properties in a form amenable to this method.

Thus, compositional methods are applied to Kendall's bCANdle framework.

One feature of the present work is that a CAN channel is modelled as an automaton, a "process" in UPPAAL terms. The method is general: it may be applied to systems that do not use CAN at all, subject to the restrictions imposed by compositionality theorems (essentially, the components do not

share variables, and communicate by discrete synchronisations). The method could be adapted to some other network technology via a time-automaton model of the process of transmitting a message.

Although the compositionality theorems have been developed specifically for time automata and for UPPAAL systems, the possibility exists of adapting the method to use other model checking tools and variations on timed-automata semantics.

1.5 Organisation of the Thesis

After an account of controller area networks and of some relevant basic modelling and model-checking concepts (chapter 2), chapter 3 surveys work in this area in more detail, particularly work which informs the present project to develop a compositional bCANdle.

The two chapters following this describe in some detail work on developing bCANdle models into an equivalent form which lends itself to a *compositional* approach. Chapter 4 recapitulates bCANdle and chapter 5 develops an equivalent formalism in terms of parallel composites of timed automata which can be modelled in UPPAAL, and shows it is semantically equivalent (bisimilar) to the "original" bCANdle modelling.

This formalism exhibits a CAN channel in a natural way as a process which shares variables with processes wishing to communicate via the channel. As will be seen in the sequel, "compositionality" theorems (i.e. which allow the component-wise inference of properties) are easiest to derive when we can presume the components do *not* share variables. It is therefore expedient to recast the rather natural modelling developed in chapter 5 into a form with no variables shared between the component automata. This is done in chapter 6 where the no-shared-variables model is shown to be equivalent to the original, and the implications of programming the models as UPPAAL systems and processes are explored.

In chapter 7 are developed compositionality theorems for timed automata which form the basis for compositional reasoning about the behaviour of broadcasting embedded systems modelled in bCANdle. Chapter 8 develops

the methodology for this via a number of examples in which we also show the benefits in terms of processor time and space requirements of compositional reasoning. This practical work involves software tools for generating UPPAAL source code from higher-level model definitions.

Chapter 9 summarises the main conclusions of this work (§9.1) and then discusses some possible future directions for research, both theoretical and practical. Possibilities exist to refine the present work by making use of some of the theoretical work surveyed in chapter 3. While the work outlined in this thesis proves the concept of a compositional model-checking method for broadcasting embedded systems, it also motivates the practical development of one or more software tools to assist it.

Appendices contain the details of proof of the compositional reformulation of bCANdle, details of `CANGen`, a tool written to assist with of the formulation of no-shared-variables models, and source code of the examples discussed in chapter 8.

Chapter 2

A Survey of Basic Concepts

This chapter outlines a number of model-checking concepts and definitions which will inform or be used in later work in the thesis.

2.1 Temporal Logics

The texts by Bérard, Bidoit, Finkel, Laroussinie, Petit, Petrucci, Schnoebelen and McKenzie [26] and by Clarke, Grumberg and Peled [45] develop the theory of timed automata in ways similar to this, although definitions vary in detail and we shall meet some other interestingly variant definitions in the sequel. They also describe a number of formal logics which can be used to express properties of automata and, through them, properties of systems.

[45], §3.1 (summarised in Clarke and Schlingloff [46]) and [26], §2.1 describe the *Computation Tree Logic* CTL*. This is a formal logic whose semantics are based on *Kripke structures*: very loosely, these are trees (acyclic directed graphs) similar to the labelled transition systems which model automata, and carrying a set of atomic *propositions* at each node. The nodes represent instants of (*discrete*) time or stages of a computation or run.

2.1.1 Syntax of CTL*

The language is based on an initial set of *atomic formulae* and features modal *temporal operators* such as $\bigcirc\varphi$ (meaning, informally, φ will be true at the

"next" instant), $\Diamond\varphi$ (φ true "now" or at some later time), $\Box\varphi$ (φ true now and at all later times), $\varphi\mathcal{U}\psi$ (ψ true now or later; meanwhile φ is true), its weak version $\varphi\mathcal{W}\psi$ (φ is true now and until ψ becomes true) and its dual $\varphi\mathcal{R}\psi$ (ψ holds until φ first becomes true). Most of these are derivable from others – e.g. $\Diamond\varphi \triangleq \texttt{true}\mathcal{U}\varphi$, $\Box\varphi \triangleq \neg\Diamond\neg\varphi$, $\varphi\mathcal{W}\psi \triangleq \varphi\mathcal{U}\psi \vee \Diamond\varphi$, $\varphi\mathcal{R}\psi \triangleq \neg(\neg\varphi\mathcal{U}\neg\psi)$.

Computation Tree Logic envisages a tree-like structure for time. From any instant, multiple alternative futures may branch out. Accordingly the syntax features *path quantifiers* $\forall$ (for all paths ...) and $\exists$ (for some path ...).

Two classes of formulae are defined thus -

- An atomic formula is a *state formula*;
- A boolean combination of state formulae, $\varphi \wedge \psi$ or $\varphi \vee \psi$ or $\neg\varphi$, is a state formula;
- A state formula is also a path formula;
- Boolean combinations of path formulae are again path formulae;
- If ρ is a path formula, $\forall\rho$ and $\exists\rho$ are *state* formulae.
- If $\rho,\ \chi$ are path formulae, $\bigcirc\rho,\ \Diamond\rho,\ \Box\rho,\ \rho\mathcal{U}\chi,\ \rho\mathcal{W}\chi,\ \rho\mathcal{R}\chi$ are *path formulae.*

2.1.2 Semantics

Branching time can be represented formally by *Kripke structures* – see, e.g., [45]. A Kripke structure for a CTL* logic is a triple $\langle S, R, L\rangle$ where S is a set of time "nodes" or "instants", and $R \subseteq S \times S$ is a *total* binary relation on S defining the "connectivity" of time instants: $(s, s') \in R$ means s' is "in the future of" s. A *path* in $\langle S, R, L\rangle$ is an infinite sequence $\langle s_0, s_1, s_2, ...\rangle$ of states such that $\forall n(s_n, s_{n+1}) \in R$. In the sequel this sequence is denoted $\overrightarrow{s}$ for short. L is a function mapping each state to the set $L(s)$ of atomic propositions true there.

One can interpret a CTL* over a Kripke structure $\langle S, R, L\rangle$. If $s \in S$ and $\overrightarrow{s} = \langle s_0, s_1, s_2, ...\rangle$ is a path in the structure, the following clauses define

recursively what it means for a state formula to be true at s and a path formula to be true on $\overrightarrow{s}$. $\overrightarrow{s}^k$ denotes the *tail* of $\overrightarrow{s}$, the sequence $\langle s_k, s_{k+1}, ...\rangle$. $\overrightarrow{s}$ is a *path from* s' iff $s_0 = s'$. α denotes an atomic formula, φ, ψ state formulae, ρ, χ path formulae.

- $s \models \alpha$ iff $\alpha \in L(s)$;
- $s \models \varphi \wedge \psi$ iff $s \models \varphi$ and $s \models \psi$; $s \models \varphi \vee \psi$ iff $s \models \varphi$ or $s \models \psi$; $s \models \neg\varphi$ iff $s \not\models \varphi$.
- $\overrightarrow{s} \models \varphi$ iff $s_0 \models \varphi$;
- $\overrightarrow{s} \models \rho \wedge \chi$ iff $\overrightarrow{s} \models \rho$ and $\overrightarrow{s} \models \chi$; $\overrightarrow{s} \models \rho \vee \chi$ iff $\overrightarrow{s} \models \rho$ or $\overrightarrow{s} \models \chi$; $\overrightarrow{s} \models \neg\rho$ iff $\overrightarrow{s} \not\models \rho$;
- $s \models \exists\rho$ iff for some path $\overrightarrow{s}$ from s, $\overrightarrow{s} \models \rho$; $s \models \forall\rho$ iff for every path $\overrightarrow{s}$ from s, $\overrightarrow{s} \models \rho$;
- $\overrightarrow{s} \models \bigcirc\rho$ iff $\overrightarrow{s}^1 \models \rho$; $\overrightarrow{s} \models \Diamond\rho$ iff $(\exists k)\overrightarrow{s}^k \models \rho$; $\overrightarrow{s} \models \Box\rho$ iff $(\forall k)\overrightarrow{s}^k \models \rho$; $\overrightarrow{s} \models \rho\mathcal{U}\chi$ iff $(\exists k)(\overrightarrow{s}^k \models \chi$ and $(\forall j < k)\overrightarrow{s}^j \models \rho)$; and analogously for $\rho\mathcal{W}\chi$, $\rho\mathcal{R}\chi$.

One can add to a Kripke structure $\langle S, R, L\rangle$ a distinguished *initial state*, s^0, and say the resulting Kripke model satisfies φ, $\langle S, s^0, R, L\rangle \models \varphi$, iff $s^0 \models \varphi$ as defined above.

Labelled Transition Systems A labelled transition system (LTS) is a structure that is slightly richer than the Kripke structure: $\langle \Sigma, \sigma^0, A, \rightarrow\rangle$. Σ is a set of *states* and σ^0 an *initial* state, A a set of *action labels* (the action *alphabet*) and $\rightarrow \subseteq \Sigma \times A \times \Sigma$ the *transition relation*. Notice the suggestive terminology: the states may represent states of some system, and the labelled transitions changes of state the system might undergo. One writes

$$\sigma_1 \xrightarrow{\lambda} \sigma_2$$

to mean $(\sigma_1, \lambda, \sigma_2) \in \rightarrow$: state σ_1 evolves into state σ_2 on undergoing an action (transition) labelled λ.

A *path* or *run* of the system is a sequence

$$\sigma_0 \stackrel{\lambda_0}{\rightarrow} \sigma_1 \stackrel{\lambda_1}{\rightarrow} \sigma_2 \stackrel{\lambda_2}{\rightarrow} ...$$

which may be infinite or may terminate at a state σ_n. The *length* of the run/path is this n. A state σ is *reachable* from σ_0 iff for some run from σ_0, for some n, $\sigma = \sigma_n$.

Usually we are interested in runs from $\sigma_0 = \sigma^0$.

If the system has properties expressible in a suitable formal language, one can add a semantic function L as above and evaluate a CTL* formula over the LTS.

Thus if σ is a state of a labelled transition system or node of a Kripke structure, $\sigma \models \forall\Box\varphi$ iff φ is true at all nodes of the subtree rooted at σ; $\sigma \models \exists \bigcirc \varphi$ iff φ is true at some child node of σ, and $\sigma \models \exists(\varphi \mathcal{U} \psi)$ iff some path $\sigma = \sigma_0, \sigma_1, ..., \sigma_k$, $\sigma_k \models \psi$ and $\sigma_i \models \varphi$ for $i < k$.

2.1.3 CTL, LTL and μ-Calculus

Actually all these examples are drawn from the sub-logic CTL in which temporal operators must be preceded immediately by $\forall$ or $\exists$. This logic is sufficiently expressive of "branching time" properties for most purposes and is (more or less) the logic used by the tool UPPAAL ([50][89]; see also the "help" bundled with the tool) for expressing properties of models built of timed automata enriched with data variables.

Essentially, CTL deals only with state formulae, which are atomic formulae; boolean combinations of state formulae; and, for state formulae φ, ψ, the formulae $\forall \bigcirc \varphi$, $\exists \bigcirc \varphi$, $\forall\Diamond\varphi$, $\exists\Diamond\varphi$, $\forall\Box\varphi$, $\exists\Box\varphi$, $\forall(\varphi \mathcal{U} \psi)$, $\exists(\varphi \mathcal{U} \psi)$, $\forall(\varphi \mathcal{R} \psi)$, $\exists(\varphi \mathcal{R} \psi)$. Combinations of $\neg$, $\vee$, $\exists\bigcirc$, $\exists\Box$ and $\exists(\mathcal{U}\_)$ suffice, for in addition to de Morgan's laws, the following equivalences can easily be checked using the semantics outlined above: $\forall \bigcirc \varphi \iff \neg\exists \bigcirc \neg\varphi$, $\forall\Diamond\varphi \iff \neg\exists\Box\neg\varphi$, $\exists\Diamond\varphi \iff \exists(\mathtt{true}\mathcal{U}\varphi)$, $\forall\Box\varphi \iff \neg\exists\Diamond\neg\varphi$, $\forall(\varphi\mathcal{U}\psi) \iff \neg\exists(\neg\psi\mathcal{U}(\neg\varphi \wedge \neg\psi)) \wedge \neg\exists\Box\neg\psi$, $\forall(\varphi\mathcal{R}\psi) \iff \neg\exists(\neg\varphi\mathcal{U}\neg\psi)$ and $\exists(\varphi\mathcal{R}\psi) \iff \neg\forall(\neg\varphi\mathcal{U}\neg\psi)$.

Defining $\forall(\_\mathcal{W}\_)$, $\exists(\_\mathcal{W}\_)$ is left as an exercise.

Another interesting sublogic of CTL* is *linear time logic*, LTL: essentially the fragment of formulae of the form $\forall\varphi$ where φ is a boolean and/or temporal combination of atomic formulae. The semantics of LTL is captured by computation paths with no branching. LTL is the logic underlying the model-checking tool SPIN and associated modelling language PROMELA developed by Gerhard Holzmann: see Holzmann [71, 72]. Alas this language and tool do not support models with explicit time in the way that timed-automata-based models do. [26] and §2.4 briefly discusses differences in expressiveness of CTL, LTL, CTL*.

As [45], §6.1 explains, each CTL operator can be defined in terms of the least or greatest *fixed point* of some monotonic *predicate transformer*. We are interested here in *predicates* on the states/nodes of some Kripke structure – subsets of the set S of all such. A mapping $f : \wp(S) \to \wp(S)$ is *monotonic* iff $X \subseteq Y \Rightarrow f(X) \subseteq f(Y)$. An X such that $f(X) = X$ is a *fixed point* of f. Because $\wp(S)$ is a *complete* lattice it turns out that that every monotonic mapping f has a *least* fixed point ($\subseteq$ every other fixed point) denoted $\mu X.f(X)$ and a *greatest* ($\supseteq$ every other fixed point) one, denoted $\nu X.f(X)$[1].

If we identify a formula φ with its "truth set", the predicate $\{\sigma \in S | \sigma \models \varphi\}$ then

- $\forall\Diamond\varphi$ "is" $\mu X.(\varphi \vee \forall \bigcirc X)$, and $\exists\Diamond\varphi$ "is" $\mu X.(\varphi \vee \exists \bigcirc X)$,
- $\forall\Box\varphi$ "is" $\nu X.(\varphi \wedge \forall \bigcirc X)$, and $\exists\Box\varphi$ "is" $\nu X.(\varphi \wedge \exists \bigcirc X)$,
- $\forall(\varphi\mathcal{U}\psi)$ "is" $\mu X.(\psi \vee (\varphi \wedge \forall \bigcirc X))$ and $\exists(\varphi\mathcal{U}\psi)$ "is" $\mu X.(\psi \vee (\varphi \wedge \exists \bigcirc X))$
- $\forall(\varphi\mathcal{R}\psi)$ "is" $\nu X.(\psi \wedge (\varphi \vee \forall \bigcirc X))$ and $\exists(\varphi\mathcal{R}\psi)$ "is" $\nu X.(\psi \wedge (\varphi \vee \exists \bigcirc X))$

and so on.

[1] If $M = \{X | f(X) \subseteq X\}$ and $m = \cap M$ and $N = \{X | X \subseteq f(X)\}$ and $n = \cup N$ then it is easy lattice algebra to infer that $M \cap N$ is precisely the set of fixed points of f; m, n are fixed points, and *any* fixed point z satisfies $m \subseteq z \subseteq n$. Thus $m = \mu X.f(X)$ and $n = \nu X.f(X)$. By more easy lattice algebra $f^k(\emptyset)$, $k = 1, 2, ...$ is monotonic increasing and converges to $\mu X.f(X)$, and $f^k(S)$, $k = 1, 2, ...$ is monotonic decreasing and converges to $\nu X.f(X)$; if S is finite, the sequences converge after finitely many steps.

Thus, these temporal logics are relatives of the *modal μ-calculus* of which a good general account is given by J Bradfield and C Stirling in [30]. More general μ-calculi include modalities of the form $[a]\varphi$, $\langle a\rangle\varphi$, where a is an action label (say, from the alphabet of some automaton). These are true at state σ iff every (some) transition labelled a leading from σ leads to a state where φ is true.

A brief account of μ-calculus is also given by F Laroussinie and K G Larsen [87], who use it with their model checking tool CMC, and by R Mateescu and M Sighireanu[95], also in a symbolic model-checking setting.

2.2 Timed Automata

The *automaton* is a computing abstraction that has been around for a long time (see, e.g., [73]) and comes in a number of variations which have in common the idea of a set L of *locations* or *states*, a set A of labelled *transitions* between states, and rules determining what transitions are possible, formulated perhaps as a relation $E \subseteq L \times A \times L$. The labelled transition systems described above provide a semantic framework for these automata, in which possible *runs* of an automaton (sequences of states interspersed with labels of a possible transition from each state) are exhibited.

When used to study real-time systems, runs (in a labelled transition system) describe possible behaviours of the system modelled by the automaton. Hypotheses about the system are formulated in a formal language such as CTL*, and their validity or satisfiability checked within the semantic framework described above.

A finite state automaton can be pictured as a *directed graph* with vertices L and edges E labelled by A. $(l, a, l') \in E$ is denoted graphically -

$$l \xrightarrow{a} l'$$

.

The automaton can be declared to have a distinguished *initial* location/state $l^0 \in L$ and perhaps also a subset $F \subseteq L$ of "desirable" *final states*.

One can enquire whether states in F are *reachable*; whether all possible runs end on a state in F (if finite) or visit F "infinitely often" (if infinite). The latter case refers to a run in whose sequence of visited states, every state l^k is followed eventually by a state $l^n \in F$, $n > k$. One speaks in this of *Büchi acceptance* of the run and from this notions of *fairness* of system behaviour have been developed – see, e.g. Clarke *et al.* [45], §3.3.

The *timed automaton* is a variation much used in the study of real-time systems where it is desired to model time more explicitly than seen so far.

2.2.1 Definition

The timed automaton can be defined in a number of ways; see, for instance, Alur and Dill [7]. To establish specifics and notation it is defined here to be a structure $(L, l^0, A, \mathcal{H}, E, I)$ where as before, L is a set of *locations*, E a set of transition *edges* and A a set of labels for *discrete actions*. A *timed* automaton has in addition a set $\mathcal{H}$ of *clock* variables. An *atomic clock constraint* is an inequality between a clock variable or a difference of two clock variables and a real time; let $\mathcal{Z}(\mathcal{H})$ denote the set of *clock zones* – conjunctions of atomic clock constraints. The transition relation E of a *timed* automaton has the form $E \subseteq L \times \mathcal{Z}(\mathcal{H}) \times A \times \wp(\mathcal{H}) \times L$: a transition from $l \in L$ to $l' \in L$ is labelled by an action $a \in A$ and also by a *guard* $\zeta \in \mathcal{Z}(H)$ which must be satisfied for the transition to be taken, and subset of clocks which will be *reset* to 0 when the transition is taken. I is a function which associates to each $l \in L$ an *invariant* $I(l) \in \mathcal{Z}(\mathcal{H})$ which must be satisfied for the automaton to "be" at location l.

An transition edge $(l, \zeta, a, H, l') \in E$ (where $l, l' \in L,\ \zeta \in \mathcal{Z}(\mathcal{H}),\ H \subseteq \mathcal{H}$) is denoted graphically -

$$l \xrightarrow{\zeta, a, H} l'$$

2.2.2 Semantics

The *semantics* of such a timed automaton is given by a *timed transition system* (cf. Alur, Dill [7], §3.4). This is a development of the labelled transition system $\langle \Sigma, \sigma^0, A, \rightarrow \rangle$ described above. The elements of Σ are *states* (and σ^0

an *initial* state) and the elements of A are transition labels as before: these are *discrete action* transitions. There are also *time passage* transitions $t \in \mathbb{R}$ ($\mathbb{R}$ unless otherwise stated denotes the *non-negative* reals) and the set of action labels is $A \cup \mathbb{R}$. We assume $A \cap \mathbb{R} = \emptyset$ and require that time passage be

- *deterministic* – if $t \in \mathbb{R}$ and $\sigma_1 \xrightarrow{t} \sigma_2$ and $\sigma_1 \xrightarrow{t} \sigma_2'$ then $\sigma_2 = \sigma_2'$;
- *additive* – if $\sigma_1 \xrightarrow{t_1+t_2} \sigma_2$ then for some state σ', $\sigma_1 \xrightarrow{t_1} \sigma' \xrightarrow{t_2} \sigma_2$.

Given a timed automaton $(L, l^0, A, \mathcal{H}, E, I)$, a timed transition system (TTS) $(\Sigma, \sigma^0, A \cup \mathbb{R}, \rightarrow)$ is defined as follows. $\Sigma = L \times \mathbb{R}^{\mathcal{H}}$: the *states* are pairs (l, v) consisting of a location l and a *clock valuation* $v : \mathcal{H} \rightarrow \mathbb{R}$, an assignment of values to the clocks. The initial state $\sigma^0 = (l^0, 0)$ (all clocks are initially set to 0); transitions are labelled by A action labels or real numbers. The transition relation $\rightarrow \subseteq \Sigma \times (A \cup \mathbb{R}) \times \Sigma$ is written $(l, v) \xrightarrow{a} (l', v')$ for $((l, v), a, (l'.v')) \in \rightarrow$, and contains all

- *discrete action transitions* $(l, v) \xrightarrow{a} (l', v[H := 0])$ where $(l, \zeta, a, H, l') \in E$ and $v \models \zeta$ and $v[H := 0] \models I(l')$. Here, $v[H := 0]$ denotes the clock valuation which assigns value 0 to all clocks in subset H and otherwise agrees with v; $v \models \zeta$ means ζ is *true* for the clock values assigned by v.
- *time-elapse transitions* $(l, v) \xrightarrow{t} (l, v + t)$ where $t \in \mathbb{R}$ and for all $t' \in [0, t] : v + t' \models I(l)$. Here, $v + t$ denotes the valuation $h \mapsto v(h) + t$.

2.2.3 Parallel Composition

To model concurrent real-time systems naturally entails the *composition* of timed automata. There are a number of ways of doing this; Bérard *et al.* [26] §1.5 give a rather general construction. A particular case used in bCANdle work [79] is as follows. Given two timed automata $\mathbb{A}_i = (L_i, l_i^0, A_i, \mathcal{H}_i, E_i, I_i), i = 1, 2$, define

$$\mathbb{A}_1 \parallel \mathbb{A}_2 = (L_1 \times L_2, (l_1^0, l_2^0), A_1 \cup A_2, \mathcal{H}_1 \cup \mathcal{H}_2, E, I)$$

where
$I(l_1, l_2) = I_1(l_1) \wedge I_2(l_2)$ and
$(l_1, l_2), \zeta, a, H, (l_1', l_2') \in E$ iff one of the following obtains for some
$(l_1, \zeta_1, a, H_1, l_1') \in E_1$ and/or some $(l_2, \zeta_2, a, H_2, l_2') \in E_2$:

- $a \in A_1 \cap A_2, \zeta = \zeta_1 \wedge \zeta_2, H = H_1 \cup H_2$. In this case a is a *synchronised* action of the two automata acting in step.
- $a \in A_1 - A_2, \zeta = \zeta_1, H = H_1, l_2 = l_2'$; or
- $a \in A_2 - A_1, \zeta = \zeta_2, H = H_2, l_1 = l_1'$. In these cases, a is an *interleaving* transition: one automaton acts while the other waits.

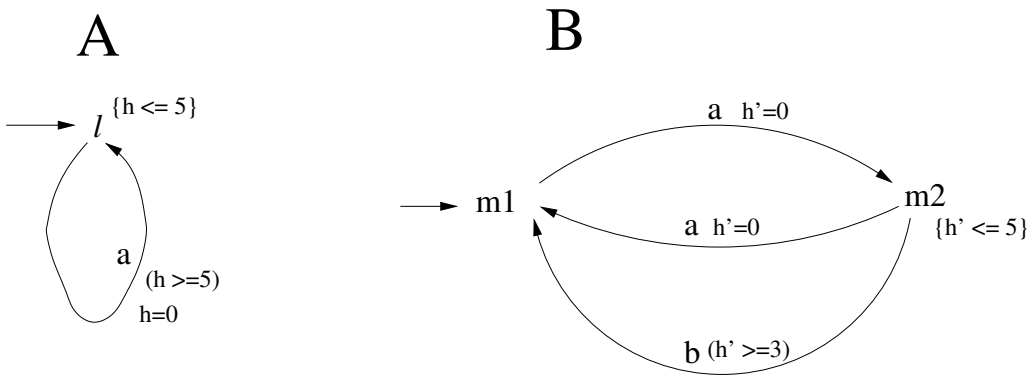


Figure 2.1: Example: two parallel Timed Automata

Example Figure 2.1 shows a simple example of a system comprising two parallel processes represented by timed automata $\mathbb{A}$, $\mathbb{B}$. $L_{\mathbb{A}} = \{l\}$, $l^0_{\mathbb{A}} = l$, $A_{\mathbb{A}} = \{a\}$, $\mathcal{H}_{\mathbb{A}} = \{h\}$, $E_{\mathbb{A}} = \{(l, \text{“}h >= 5\text{”}, a, \{h\}, l)\}$, $I_{\mathbb{A}}(l) = \text{“}h <= 5\text{”}$. $L_{\mathbb{B}} = \{m_1, m_2\}$, $l^0_{\mathbb{B}} = m_1$, etc. $I_{\mathbb{B}}(m_1) = \texttt{true}$, $I_{\mathbb{B}}(m_2) = \text{“}h' <= 5\text{”}$.

In the product system, the two components perform an a-action every 5 time units, in step: the logic of $\mathbb{A}$ forces this. $\mathbb{B}$ may (non-deterministically) move by a b-action from location m_2 to m_1 3 to 5 time units after arriving at m_2: so the a-action in $\mathbb{B}$ is non-deterministically one of the edges depicted.

Parallel Product of Several Timed Automata This construction generalises to any number of component automata. Given a set of timed automata $\mathbb{A}_i = (L_i, l_i^0, A_i, \mathcal{H}_i, E_i, I_i), i = 1...n$, where the components in the parentheses are respectively the location set, the initial location, the action alphabet, the set of clocks, the edge (transition) relation and the invariant function, the *product* is $\|_{i=1}^n \mathbb{A}_i = \mathbb{A}_1 \| ... \| \mathbb{A}_n = (L, l^0, A, \mathcal{H}, E, I)$ where

$$\begin{aligned} L &= L_1 \times ... \times L_n \\ l^0 &= (l_1^0, ..., l_n^0) \\ A &= A_1 \cup ... \cup A_n \\ \mathcal{H} &= \mathcal{H}_1 \cup ... \cup \mathcal{H}_n \\ I(l_1, ..., l_n) &= I_1(l_1) \wedge ... \wedge I_n(l_n) \end{aligned}$$

... and E has, for $\zeta \in \mathcal{Z}(\mathcal{H}), a \in A, H \subseteq \mathcal{H}$,

$$(l_1, ..., l_n) \xrightarrow{\zeta, a, H} (l_1', ..., l_n')$$

iff *either* $(\exists k)(\forall i \neq k) l_i = l_i'$ and $l_k \xrightarrow{\zeta, a, H} l_k'$
or $(\forall i) l_i \neq l_i' \Rightarrow l_i \xrightarrow{\zeta_i, a, H_i} l_i'$ where $\zeta = \wedge_{i=1}^n \zeta_i$ and $H = \cup_{i=1}^n H_i$.

The first of these disjuncts is an action on a single component while nothing happens on the other components, while the second is a synchronised action on several components.

2.2.4 An Extended Timed Automaton

It will be useful in the sequel to have a slightly extended timed automaton construction.

- In addition to *clocks*, the automaton may have *variables* of various data types – boolean, integer, real, including arrays and structures.
- *Guards* on transitions may include in their conjunctions equality and inequality tests on variables as well as clocks;
- The *clock reset* set associated with a transition becomes a *set of updates*

which includes clock resets as well as updates of other variables, such as setting a variable to a value, incrementing or decrementing a variable.

- The semantics extend in a natural way: a state of the timed transition system includes a valuation of all the variables, not just the clocks (although only the clocks adjust their values in a "time passage" transition).

- The parallel product likewise extends naturally: an action transition in the product involves a conjunction of the guards on the component transitions when two or more synchronise, and an update set which is the union of the component update sets.

It is possible for the product to have variables shared between the components. The model-checking tool UPPAAL [50][89] supports extended timed automata like this, offering a C++-like syntax for updates, subrange types as well as a full integer type, arrays and (most recently) structured data types. Some restrictions are placed on the ways clock variables may figure in updates, but simple resets are always permitted.

UPPAAL offers particular kinds of synchronised action which will turn out to be useful to us. *Binary* synchronisation involves just two components (UPPAAL defines one of these as "offering" the synchronisation, the other as "receiving" it). This synchronisation may be defined as "urgent", meaning that the action occurs as soon as the semantics permit: no time delay is allowed even if state invariants permit.

UPPAAL *broadcast* synchronisation involves two or more components: again one is deemed to be offering" the synchronisation; the other components receiving it. The sender may act as soon as it is able; receiving components synchronise as defined formally above §2.2.3 *if* they are ready. The synchronised action occurs *as soon as* the sender is ready.

These synchronisation types are discussed further in the sequel (§5.3).

2.3 Model Checking

The basic endeavour is, having formulated a model of some system as some sort of automaton, to check by a search of its state space whether some property expressed in a temporal logic or μ-calculus is satisfied by some run, or by all runs. Interesting special cases are: whether or not a state or set of states is *reachable* (a safety property) or whether, say, all runs are bound to visit a state in the set (a "liveness" property). One might check a "fairness" property – whether all runs visit a state (or perform an action), infinitely often in the sense that, where $\sigma_0, \sigma_1, \sigma_2, ..., \sigma_n, ...$ is the sequence of the states in a run, $\forall n \exists k > n : \sigma_k$ is in the set.

The state space will in general be very large and so will be searched by a software tool. UPPAAL [50][89] and CMC [87] work in this way and texts by Clarke *et al.* [45], Clarke and Schlingloff [46] and Bérard *et al.* [26] describe ways of doing this. The remainder of this section will briefly review the main algorithms and data structures employed.

2.3.1 Algorithms for Model-checking CTL*, CTL, LTL

Clark *et al.* [45] describe an algorithm for checking a CTL state formula over a Kripke structure $\langle S, R, L \rangle$. In brief, suppose the formula φ to be checked is expressed entirely in terms of $\neg, \vee, \exists\bigcirc, \exists(\_\mathcal{U}\_), \exists\Diamond$ (since every CTL formula is equivalent to one built of these connectives). For each $s \in S$ one builds a set $lbl(s)$ of sub-formulae of φ as follows.

Initially, $lbl(s) = L(s)$.

Then, the algorithm adds sub-formulae of φ to the sets $lbl(s)$, $s \in S$, by proceeding recursively on the complexity of the sub-formulae using the following cases:

- For each $s \in S$, if $\neg\psi$ is a sub-formula and $\psi \notin lbl(s)$, add $\neg\psi$ to $lbl(s)$;
- If $\psi \vee \theta$ is a sub-formula and $\psi \in lbl(s)$ or $\theta \in lbl(s)$, add $\psi \vee \theta$ to $lbl(s)$;
- If $\exists \bigcirc \psi$ is a sub-formula and $\psi \in lbl(s')$ and $(s.s') \in R$, add $\exists \bigcirc \psi$ to $lbl(s)$;

- To decide whether to add a sub-formula $\exists(\psi \mathcal{U} \theta)$ to each $lbl(s)$: first determine the set $T = \{s \in S | \theta \in lbl(s)\}$; then add $\exists(\psi \mathcal{U} \theta)$ to $lbl(s')$ for all s' such that $s' = s_0 R s_1 R ... R s_n \in T$ and $\psi \in lbl(s_0) \cap ... \cap lbl(s_n)$. Very neat pseudo-code for this is given in [45] §4.1.

- To decide whether to add a sub-formula $\exists\Box\psi$ to each $lbl(s)$, first, compute $S' = \{s \in S | \psi \in lbl(s)\}$ and let R', L' be the restrictions to S' of R, L, and let T be the set of nodes which lie in a non-trivial *strongly connected component* of the graph (S', R')[2]. The construction of T has algorithmic time complexity of order $|S'| + |R'|$. Add $\exists\Box\psi$ to $lbl(s')$ for all s' such that $s' = s_0 R' s_1 R' ... R' s_n \in T$. Again this is neatly pseudocoded in [45] §4.1.

At the end of the algorithm, φ holds at those $s \in S$ for which $\varphi \in lbl(s)$. The subalgorithms of the last two bullet points each have time complexity $O(|S| + |R|)$, so the complexity of checking algorithm is of the order of the size of $\varphi \times$ this.

It is CTL that will be of most interest to us in the sequel; however both [45] and [26] discuss model-checking LTL and CTL* briefly. The former describes an algorithm for constructing, given an LTL (path) formula ρ and a Kripke structure $\langle S, R, L \rangle$, a *tableau* against which (for $s \in S$) $s \models \exists\rho$ can be checked. This is a directed graph each node of which is an *atom*, (s, K), $s \in S$ and K a maximal consistent set of formulae whose truth values can influence that of ρ and which are also consistent with $L(s)$. $s \models \exists\rho$ iff there exists a node (s, K) with $\rho \in K$ and a path into a certain strongly connected component of the graph. The computational complexity of checking this is linear in the size of the Kripke model but exponential in the size of ρ. The tableau idea and the use of maximal consistent set s of formulae is reminiscent of the methods in mathematical logic of canonical model construction.

[45] goes on to outline briefly an extension of this tableau method to a method for checking CTL* formulae: ρ is permitted to have arbitrary state sub-formulae, not just atomic ones, and a multi-stage check working

[2] a $\subseteq$-maximal R'-path-connected subgraph, excluding *trivial* ones with just one node without a self-loop

recursively over the complexity of state sub-formulae (cf. the CTL algorithm described above) is done. This combination of the two approaches has the same complexity as the LTL checking algorithm.

2.3.2 Symbolic Model Checking

The CTL checking algorithm seen above has complexity linear in the size of the transition graph and in the size of the formula. A problem is that the graph (the transition system) may "explode" in size.

In *symbolic model checking* approaches, a large number of states that need not be distinguished for the purpose of the property being checked are represented by a single symbol. [45], chapter 6 and [26], chapter 4 describe a method of symbolic model checking in which the symbols are the boolean formulae at the nodes of a Kripke structure.

Ordered binary decision diagrams are used to represent boolean functions and by extension other set-theoretic entities, including Kripke structures and formulae, very compactly, and operate on them and check them for equivalence very efficiently.

By *boolean* function we mean a function $\mathbf{2}^n \to \mathbf{2}$ ($\mathbf{2} \triangleq \{0,1\}$) for some $n \in \mathbb{N}$. A *binary decision diagram* (BDD) for a boolean function is an acyclic directed graph, a tree, which has vertices of two types:
- a *terminal* vertex is labelled with a *value* 0 or 1 and has no edges pointing to any "child" vertices;
- a *non-terminal* is labelled with an "independent variable" to the function, and has two edges to child vertices, labelled 0, 1.

Suppose the independent variables are $\{x_1, ..., x_n\}$. The (sub)tree rooted at vertex v generates a function $f_v(x_1, ..., x_n)$ defined recursively as follows.
- If v is terminal, return its value;
- Otherwise let x_i denote the variable labelling v, and say $v \xrightarrow{0} v_0$, $v \xrightarrow{1} v_1$ [v_0, v_1 are the roots of the two sub-trees pointed to by v] and suppose the return values of f_{v_0} and f_{v_1} are determined already; return $(\neg x_i \wedge f_{v_0}(x_1, ..., x_n)) \vee (x_i \wedge f_{v_1}(x_1, ..., x_n))$.

In this way a BDD determines a boolean function. A boolean function corresponds to several BDDs in general, but it turns out (Clarke *et al.*[45], Bryant [39] [40]) that that a boolean function corresponds to just one BDD in *canonical form* in which the variables labelling non-terminal nodes appear in the same order along each path from the root, and there are no isomorphic or redundant sub-trees. An *ordered* binary decision diagram, OBBD, is made, given a strict total ordering of the variables, by first building a BDD with nodes in depth-order, than transforming it by (1) eliminating all but one terminal node labelled 0 and all but one labelled 1, and redirecting edges to these; (2) whenever two non-terminal nodes have the same variable label and the same children, eliminate the duplicate(s) and redirect incoming edges to the remaining one; (3) whenever a node has child 0 = child 1, eliminate the node and redirect its incoming edges to the child node. The emphReduce algorithm of Bryant applies (1-3) repeatedly until no further reduction in the size of the graph results.

The OBDD provides, in general, a compact representation of boolean functions, although the size of the graph can vary greatly with the ordering of variables [45] and finding the optimal ordering is an NP-complete problem (Bryant [41]).

Boolean operations between boolean functions are efficiently implemented using OBDD too. If $\bullet$ denotes one of the 16 possible two-argument operations, $f \bullet f'$ is computed as follows (Bryant [39]). Let v, v' denote the root nodes of OBBDs for $f.f'$, and x, x' the variables labeling v, v'. If $x \equiv x'$ the equation

$$f \bullet f' = (\neg x \wedge (f|_{x \leftarrow 0} \bullet f'|_{x \leftarrow 0})) \vee (x \wedge (f|_{x \leftarrow 1} \bullet f'|_{x \leftarrow 1}))$$

where $f|_{x \leftarrow b}$ is the function resulting from substitution of the constant bit b for variable x in f, breaks the calculation recursively into two sub-problems computable by smaller OBDD. If $x < x'$ in the canonical ordering of variables, $f'|_{x \leftarrow b} = f'$, so the equation $f \bullet f' = (\neg x \wedge (f|_{x \leftarrow 0} \bullet f')) \vee (x \wedge (f|_{x \leftarrow 1} \bullet f'))$ serves, and the case $x > x'$ is symmetrical to this. At the base of the recursion are OBDD consisting of just a terminal root node, in which case the associated

value is returned.

A naive implementation of this recursion would have exponential complexity, but by caching results of sub-computations (subgraphs of OBDD) the complexity can be kept to a linear function of the sizes of the OBDD for f, f'. Of course, some boolean operations are simple as OBDD operations: for instance negation is just a matter of interchanging the 0- and 1-terminal nodes.

Representing Kripke Structures by OBBD. Clarke *et al.* [45] suggest doing this as follows. Let $\langle S, R, L\rangle$ be a Kripke structure and assume without loss of generality that S has 2^m states. Encode these as boolean vectors $\overrightarrow{b} = (b_1...b_m)$. Then S may be represented by the OBBD of its *characteristic function* $\chi_S : \mathbf{2}^m \to \mathbf{2}$, and similarly R by the OBBD of its characteristic function $\chi_R : \mathbf{2}^m \times \mathbf{2}^m = \mathbf{2}^{2m} \to \mathbf{2}$. L, the mapping of states to the sets of atomic propositions which hold there, may equivalently be as a mapping of each atomic proposition to the set of states where it holds. We can represent this by, for each atomic proposition, an OBBD for the (characteristic function of) the set of states.

Symbolic Model Checking with OBDD. Suppose a Kripke structure represented by OBDD as explained above. Sets of states are represented by OBBD of boolean functions $\mathbf{2}^m \to \mathbf{2}$, and each such boolean function may also be thought of as a *formula*. The fundamental idea of *symbolic* model checking is to reason about or compute with symbols (formulae) or sets of states rather than individual states. To this end, a suitable calculus of these formulae/boolean functions/sets of states is useful. [45] suggests a calculus of *quantified boolean formulae* (QBF) for this purpose.

One starts with a set of m propositional variables $V = \{v_1, ..., v_m\}$ corresponding to the independent variables of the boolean functions. A QBF formula is a propositional variable; or $\neg f$ or $f \vee g$ or $f \wedge g$ where f, g are QBF formulae; or $\exists v f$ or $\forall v f$ where v is a propositional variable and f a QBF formula.

Truth assignments are elements $\sigma \in \mathbf{2}^V$. If $b \in \mathbf{2}$ is a bit, $\sigma[v \leftarrow b]$ is the

truth assignment like σ except $v \mapsto b$. For truth assignments σ and QBF formulae f a relation $\sigma \models f$ is defined recursively by:
– $\sigma \models v$ iff $\sigma(v) = 1$;
– $\sigma \models \neg f$ iff $\sigma \not\models f$;
– $\sigma \models f \vee g$ iff $\sigma \models f$ *or* $\sigma \models g$; $\sigma \models f \wedge g$ iff $\sigma \models f$ *and* $\sigma \models g$;
– $\sigma \models \exists v f$ iff $\sigma[v \leftarrow 0] \models f$ *or* $\sigma[v \leftarrow 1] \models f$;
– $\sigma \models \forall v f$ iff $\sigma[v \leftarrow 0] \models f$ *and* $\sigma[v \leftarrow 1] \models f$.

With the aid of this calculus, an algorithm for computing the (OBDD of the characteristic boolean function of the) set of states of the Kripke structure where a CTL formula is valid can be defined recursively, remembering that a CTL formula can be considered atomic or $\neg\varphi$ or $\varphi \wedge \psi$ or $\exists \bigcirc \varphi$ or $\exists(\varphi \mathcal{U} \psi)$ or $\exists\Box\varphi$, where φ, ψ are CTL formulae.

- If a is an atomic formula, the set of states satisfying a is given directly by the specification of the Kripke structure.

- The (OBDD of) the set of states satisfying $\neg\varphi$, or $\varphi \wedge \psi$, is computed from the (OBDDs of) the sets of states satisfying φ, ψ using the OBDD algorithms for computing meets and complements of boolean functions.

- To compute the set of states satisfying $\exists\bigcirc\varphi$ given $f(\overrightarrow{v})$, the characteristic function of the truth set of φ, compute the OBBD of $\exists\overrightarrow{v'}(f(\overrightarrow{v'}) \wedge R(\overrightarrow{v}, \overrightarrow{v'}))$.

- To compute the set of states satisfying $\exists(\varphi \mathcal{U} \psi)$ (given the truth sets of φ, ψ), use the fixed-point representation introduced in §2.1.3 of the set of states satisfying the formula. This is $\mu X.(\psi \vee (\varphi \wedge \exists \bigcirc X))$ where we read φ, ψ as the (OBDD for the) boolean characteristic functions of the truth sets of the respective CTL formulae. Since the set of states is finite, the minimum fixed point of any monotonic function f on subsets of states (and, in particular, that implied by $X \mapsto \psi \vee (\varphi \wedge \exists \bigcirc X)$) is the limit of the monotonic increasing sequence of iterates of f on $\emptyset$: $\bigcup_{k=1}^{\infty} f^k(\emptyset)$, and this is computable in a finite number of iterations as the $f^k(\emptyset)$ agree from some k_0 on. Clark *et al.* [45] give an algorithm

for this (§6.1), and of course f is computed via the OBDD boolean function algorithms of Bryant.

- To compute the set of states satisfying $\exists\Box\varphi$ (given the truth set of φ), proceed as in the previous case, starting from the fixed-point representation introduced in §2.1.3 for the truth set of the formula: $\nu X.(\varphi \wedge \exists \bigcirc X)$ where φ is understood as the (OBDD for the) boolean characteristic functions of the truth set of the CTL formula φ. Again, since S is finite, the maximum fixed point of this monotonic function f is the limit of the decreasing sequence $f^k(S)$: $\bigcap_{k=1}^{\infty} f^k(S)$ which is similarly to the previous case computable in finitely many steps – again, see [45] §6.1.

In view of the representation of temporal operators as fixed points of monotonic predicate transformers used in the last two items, this approach to model-checking extends in a natural way to μ-calculus formulae, and [45], chapter 7 develops this.

2.3.3 Automata and Model Checking

The constructs described so far for Kripke structures apply, of course, equally well to automata, and Clark *et al.* show how an automaton can be made corresponding to a Kripke structure by labelling edges with the propositions true at their target locations. Alternatively, and this is the way UPPAAL [50][89] works, for example, one can continue simply to attach atomic propositions to automaton locations.

Bérard *et al.* ([26] chapter 4) develop symbolic model checking in a way similar to the ideas above, but in terms of automata. In the sequel, our system models will be essentially automata, in particular timed automata.

In these terms, the basic model checking problem takes one of the forms:

- Reachability properties: $\exists\Diamond\varphi$ – can the system reach a state where φ is the case?

- Safety properties: The dual of the above: $\forall\Box\neg\varphi$ – The (generally undesirable) situation described by φ can *never* occur. A variant is

something like $\forall\Box\neg\varphi\mathcal{W}\psi$; the situation φ cannot occur until ψ has occurred.

- Liveness properties: some desirable situation φ will eventually occur: $\forall\Diamond\varphi$ or $\forall(\psi\mathcal{U}\varphi)$.

- Freedom from deadlock - a special property meaning that the system will never be in a state from which no progress is possible: $\forall\Box\neg$`deadlock`. Think of this as a kind of safety property.

Related to liveness is *bounded liveness* where one asks whether the desirable situation will occur within a given time bound, and *fairness*. One can, for a Kripke structure or transition system, define a collection F of sets of states and call an (infinite) run $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow ...$ *fair* iff for every $P \in F$ there is a state $s \in P$ which occurs infinitely often in the run: $\forall m \exists n.s_n = s$. Intuitively each P in F is a set of states which we would hope to be realised infinitely often in any run. This generalises the idea of the *Büchi* automaton which has designated a "final" location (or set of final locations) which we expect to be visited infinitely often in any run.

These may be tricky to express in CTL, but we shall look at case studies in the sequel where models are constructed in order to examine fairness.

We are often interested in the question of whether a system model meets a *specification*. A common approach is to express the model as an automaton, the specification as another automaton, or rather its *complement*, modelling *failure* to meet the specification, and model-check the composition of this with the system model in order to see what pathological behaviours are possible. In this way meeting of specification becomes a kind of reachability problem.

2.3.4 Partial-order Reduction

This is family of techniques which aim to reduce the size of the state space search tree by exploiting the fact that two or more paths of execution may be indistinguishable as far as the property being checked is concerned: there is an equivalence relation among behaviours which is a congruence as far as

exhibiting the property is concerned – see, e.g., [45], chapter 10. The search tree need contain only one member each equivalence class.

For the sake of definiteness, [45] works with execution paths in a labelled transition system $\langle S, T, S^0, L\rangle$ where S is a set of states, $S^0 \subseteq S$ a subset of initial states, L a function assigning each state a set $L(s)$ of atomic propositions true there. Thus far, we have a Kripke structure, except that T is not a single relation $\subseteq S \times S$, but rather a set of relations: $T \subseteq \wp(S \times S)$. This yields in effect a labelled transition system in which the elements $\alpha \in T$ function as labels: $s \xrightarrow{\alpha} s'$ iff $(s, s') \in \alpha$.

An execution *path* is a finite or infinite sequence $s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} ....$

Model-checking some property φ consists of constructing the transition system as a tree which will be depth-first searched for a state where φ is satisfied. A standard recursive algorithm will do this, using, for each state s, the set $\{\alpha \in T | \exists s'.(s, s') \in \alpha\}$ of (labelled) transitions *enabled* at s; except that in partial-order reduction, we construct the tree using a (hopefully) small subset (which [45] denotes $ample(s)$) of the enabled set at s, leaving out the transitions not distinguished by φ.

One way of doing this is explored by [45], assuming the transitions are deterministic: $s \xrightarrow{\alpha} s'$ for at most one s', denoted $\alpha(s)$. This is to construct, given φ, a relation $I \subseteq T \times T$ which is symmetric, irreflexive and has the properties that whenever $(\alpha, \beta) \in I$ and $\alpha,\ \beta$ are enabled at s, α is enabled at $\beta(s)$ (and by symmetry, *vice versa*), and $\alpha(\beta(s)) = \beta(\alpha(s))$. This expresses the notion that α and β are "independent" actions which commute with one another. The idea is that clearly if $(\alpha, \beta) \notin I$, they are "dependent" and one could not be in $ample(s)$ without the other; but hopefully, if $\alpha \in ample(s)$ and $(\alpha, \beta) \in I$, β can be omitted from $ample(s)$.

There are problems with this: for instance, one may have $s \xrightarrow{\alpha} s_1 \xrightarrow{\beta} s'$ and $s \xrightarrow{\beta} s_2 \xrightarrow{\alpha} s'$ with $(\alpha, \beta) \in I$, but φ might be sensitive to the choice of s_1 or s_2; or s_1, s_2 may have other successors besides s' which matter to φ.

One can define a transition α to be *invisible* with respect to a set A of atomic propositions iff $\forall s.L(s) \cap A = L(\alpha(s)) \cap A$, and two infinite paths $s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} ...$, $t_0 \xrightarrow{\beta_0} t_1 \xrightarrow{\beta_1} t_2 \xrightarrow{\beta_2} ...$ to be *stuttering equivalent* if they can each be divided into blocks of finite length such that for all k, all the states

in the k^{th} blocks of *both* paths have the same proposition labels $L(s)$. The blocks need not be of equal length.

One can define two transition systems on a state set S to be stuttering equivalent iff they have the same initial states, and for any path in one system from an initial state, there is a stuttering equivalent path from the same initial state in the other system. One is then led to properties that are *invariant under stuttering*; [45] explores ways of tightening the procedure for discarding transitions from $ample(s)$ making sure that only "invisible" states and stuttering equivalent paths are "lost". This is explored in particular for LTL properties which are invariant under stuttering.

2.3.5 Modelling and Model Checking with Timed Automata

For systems with continuous real time, *timed automata* (§2.2.1) provide useful models: their semantics (§2.2.2) give rise to *timed* transition systems, extensions of the transition systems considered in the foregoing sections. There is a problem, however: there are infinitely many clock valuations if clock values are unbounded or if time is continuous. One solution is *clock regions*. An equivalence relation is defined between clock valuations: $v \simeq v'$ iff

1. for every clock h, if c_h is the largest value h is compared with in a guard or invariant, either $v(h), v'(h)$ both $\geq c_h$; or $\lfloor v(h) \rfloor = \lfloor v'(h) \rfloor'$;

2. for every two clocks h, k such that $v(h) \leq c_h,\ v(k) \leq c_k$, v, v' agree on ordering of fractional parts: $\{v(h)\} \leq \{v(k)\} \Leftrightarrow \{v'(h)\} \leq \{v'(k)\}$;

3. for every clock h such that $v(h) \leq c_h$, $v(h)$ is integral iff $v'(h)$ is.

This relation is stable with respect to
– adding a constant: if $v \simeq v'$ then $v + t \simeq v' + t$,
– clock reset: if $v \simeq v'$ then $v[h := 0] \simeq v'[h := 0]$, and
– satisfaction of clock constraints: if $v \simeq v'$ and $v \models \zeta$ then $v' \models \zeta$.

Further, if, in the timed transition system, state (s_1, v_1) evolves to state (s_2, v_2) by a time delay followed by an action of the underlying automaton,

and $v_1 \simeq v_1'$ then there exists $v_2' \simeq v_2$ such that also state (s_1, v_1') evolves to state (s_2, v_2') by a time delay followed by an action.

Now the the state space divides into finitely many *clock regions*, $\simeq$-equivalence classes; a state is a location v together with a clock region. [26], §5.5 and [45], §17.4 describe this approach and the latter shows that,

- The graph of clock regions, considered as a transition system, is *bisimilar* or *strongly equivalent* (see §2.4 below) to the original semantic transition system;
- Where H is the set of clock variables, the number of regions is $\leq |H|!.2^{|H|}.\prod_{h\in H}(2c_h + 2)$.

This approach was formalised by R Alur, C Courcoubetis and D Dill [16][17].

Another approach employs *clock zones* – recall the set $\mathcal{Z}(\mathcal{H})$ from subsection 2.2.1. These we think of as sets of clock valuations which have a symbolic representation as conjunctions of atomic clock constraints: identify such a conjunction ζ with a set of valuations $\{v | v \models \zeta\}$. Clarke *et al.* [45], §17.5,6 shows how these provide a useful symbolic state space, with operations on clock valuations "lifting" to clock zones $\zeta \in \mathcal{Z}(\mathcal{H})$:

- $\zeta + t \triangleq \{v + t | v \models \zeta\}$ where $t \in \mathbb{R}$,
- $\zeta[H := 0] \triangleq \{v[H := 0] | v \models \zeta\}$ where $H \subseteq \mathcal{H}$.

The right hand sides of these definitions need to be shown to be actual clock zones: symbolically representable by conjunctions of atomic constraints. [45] does this by first showing any zone ζ (formula) on clocks $x_1, ...x_n$ has the form

$$(x_0 = 0) \wedge \bigwedge_{0\leq i\neq j\leq n} (x_i - x_j \prec_{ij} c_{ij}) \tag{2.1}$$

where each $\prec_{ij}$ is $\leq$ or $<$, and x_0 is a special extra clock "always zero", giving a tidy uniform notation for the two types of clock constraint.

Then the formula $\exists x_n \zeta$ is equivalent to

$$(x_0 = 0) \wedge \bigwedge_{0\leq i\neq j<n} (x_i - x_j \prec_{ij} c_{ij}) \wedge \bigwedge_{0\leq i\neq j<n} (x_i - x_j \prec_{ij} c_{in} + c_{nj})$$

From this it follows that for any clock zone ζ on clock variable x (among others), $\exists x\zeta$ is a well–defined clock zone. The definition of $\zeta[x := 0]$ is equivalent to $\exists x\zeta \wedge x = 0$ and $\zeta[H := 0]$ is a conjunction of such.

Similarly, the set of valuations $v + t$ for some $t \geq 0,\ v \models \zeta$ can be shown to be symbolically representable, and hence the definition of $\zeta + t$ is sound.

From this, we get a finite *zone graph* equivalent to the original transition system. This finite state space is explored by means of an algorithm of the form (see, e.g., Larsen *et al* [88]) -

```
PASSED:=∅
WAITING:=(l^0, ζ^0)
repeat
  begin
  get (l, ζ) from WAITING
  if (l, ζ) ⊨ φ then return "yes"
  else if ζ ⊈ ζ' for all (l, ζ') ∈ PASSED then
    begin
    add (l, ζ) to PASSED
    for all (l', ζ') which are successors to (l, ζ), ζ' ≠ ∅,
      put (l', ζ') to WAITING
    end
  end
until WAITING=∅
return "no"
```

The algorithm explores the space of clock zones by representing them as data structures called *difference bound matrices*, devised by D Dill [51]. If there are k clocks, a difference-bound matrix is a $(k + 1) \times (k + 1)$ matrix D in which $D_{0i}(D_{i0})$ gives the lower (upper) bound on clock h_i, and D_{ij} the upper bound on $h_i - h_j$. Each entry is the symbol ∞, meaning no bound, or a pair of data – a number and an indication of whether the bound is strict or non-strict. The syntactic counterpart of this is, of course, the formula (2.1) above, in which the (c_{ij}) correspond to the entries D_{ij} of the difference-bound

matrix.

Larsen *et al.* [88] develop a way of doing this efficiently by representing a difference-bound matrix as a directed graph. The clocks are vertices and the weight of an edge from x_i to x_j is the (ij)-entry of the matrix (these authors consider comparison only of type $\leq$ or $\geq$). They then develop efficient algorithms for minimizing the *size* of a graph required to represent a DBM and hence a clock zone, and also minimizing the *number* of symbolic states (l, ζ) needing to be stored in the set `PASSED`. This in turn leads to efficient (in time and in memory requirement, at worst $O(n^3)$ for n clocks) ways of determining when $\zeta \not\subseteq \zeta'$ and when $\zeta' \neq \emptyset$ in the algorithm above. These predicates, of course, are essential for ensuring the algorithm terminates, and that the `PASSED` set is correctly maintained so as ensure successor states are not unnecessarily explored multiple times. Larsen *et al.* have implemented their efficient zone-based symbolic model checking over their extended timed automata – cf. §2.2.4 above.

2.4 Simulation and Bisimulation

This section develops the definitions needed for determining when two transition systems are in some sense equivalent, or when one system's behaviour is subsumed or *simulated* by another.

2.4.1 Strong Bisimulation and Strong Equivalence

These apply to labelled transition systems $(S, A, \rightarrow)$ consisting of a set S of states, a set A of transition or action labels, and a transition relation $\rightarrow \subseteq S \times A \times S$.

Milner [98] defines a *strong bisimulation* on the system to be a relation $R \subseteq S \times S$ such that whenever $(s_1, s_2) \in R$, for all $a \in A$,

- $s_1 \xrightarrow{a} s_1'$ implies $(\exists s_2')(s_1', s_2') \in R$ and $s_2 \xrightarrow{a} s_2'$; and
- $s_2 \xrightarrow{a} s_2'$ implies $(\exists s_1')(s_1', s_2') \in R$ and $s_1 \xrightarrow{a} s_1'$

It is routine to check that the diagonal (identity) relation on S is a strong bisimulation, and that the converse, the composite and any union of strong bisimulations is a strong bisimulation. In particular, the union of all strong bisimulations is a strong bisimulation, the maximal or "largest" strong bisimulation on S, and this is an equivalence relation on S, termed *strong equivalence*: $s_1 \simeq s_2$ iff for some strong bisimulation R, $(s_1, s_2) \in R$.

A *strong bisimulation up to* $\simeq$ is defined as above, but with the composite $\simeq R \simeq$ in lieu of R in the two bulleted conditions. Milner shows straightforwardly that $s_1 \simeq s_2$ iff for some strong bisimulation up to $\simeq$, R, $(s_1, s_2) \in R$.

Intuitively, $s_1 \simeq s_2$ means that there is a one to one correspondence between runs $a_0, a_1, a_2, ...$ from s_1 and runs (with the same sequence of action labels) from s_2.

The definitions make sense when s_1 and s_2 are from two transition systems $(S_1, A, \rightarrow_1)$, $(S_2, A, \rightarrow_2)$ with a common action label alphabet. One then defines the transition systems to be strongly equivalent,

$$(S_1, S_1^0, A, \rightarrow_1) \simeq (S_2, S_2^0, A, \rightarrow_2)$$

iff their initial states are strongly equivalent – to be precise, there is a strong bisimulation $R \subseteq S_1 \times S_2$ such that $\forall s_1^0 \in S_1^0 . \exists s_2^0 \in S_2^0 . (s_1^0, s_2^0) \in R$ and $\forall s_2^0 \in S_2^0 . \exists s_1^0 \in S_1^0 . (s_1^0, s_2^0) \in R$, where S_1^0, S_2^0 denote the respective initial state subsets.

The labelled transition systems considered here include the Kripke structures, generalised Kripke structures of §2.3.4, and timed transition systems (§2.2.2).

In structures in which states s are labelled by sets $L(s)$ of atomic propositions, we require additionally of a strong bisimulation R that $R(s_1, s_2) \Rightarrow L(s_1) = L(s_2)$.

Clarke *et al.* [45] show that, given two strongly equivalent structures, corresponding states satisfy the same CTL* state formulae, and corresponding paths (i.e. whose states correspond term-wise under the bisimulation) satisfy the same CTL* path formulae.

2.4.2 Simulation

A simulation is a "one-sided" version of a bisimulation: a binary relation R between states such that whenever $(s_1, s_2) \in R$, for all $a \in A$, $s_1 \xrightarrow{a} s_1'$ implies $(\exists s_2')(s_1', s_2') \in R$ and $s_2 \xrightarrow{a} s_2'$.

This is most useful as a relation between structures or transition systems $(S_1, S_1^0, A, \rightarrow_1)$, $(S_2, S_2^0, A, \rightarrow_2)$ with initial state sets S_i^0 and a common action label alphabet. One defines that system 2 *simulates* system 1,

$$(S_1, S_1^0, A, \rightarrow_1) \preceq (S_2, S_2^0, A, \rightarrow_2)$$

iff there is a simulation $R \subseteq S_1 \times S_2$ such that $\forall s_1^0 \in S_1^0 . \exists s_2^0 \in S_2^0 . (s_1^0, s_2^0) \in R$.

In systems which label states with sets of atomic propositions, for a simulation $R \subseteq S_1 \times S_2$ from $(S_1, S_1^0, A, \rightarrow_1, L_1)$ to $(S_2, S_2^0, A, \rightarrow_2, L_2)$ we require that the atomic propositions of system 1 *include* the atomic propositions of system 2, and that whenever $(s_1, s_2) \in R$,

- The restriction of $L_1(s_1)$ to the atomic propositions of system 2 is precisely $L_2(s_2)$;
- $s_1 \xrightarrow{a} s_1'$ implies $(\exists s_2')(s_1', s_2') \in R$ and $s_2 \xrightarrow{a} s_2'$.

Then, again, system 2 *simulates* system 1,
$(S_1, S_1^0, A, \rightarrow_1) \preceq (S_2, S_2^0, A, \rightarrow_2)$ iff there is a simulation R (in the stronger sense) such that $\forall s_1^0 \in S_1^0 . \exists s_2^0 \in S_2^0 . (s_1^0, s_2^0) \in R$.

Note that as a relation between transition systems, $\preceq$ is reflexive and transitive, a *preorder*.

Clarke *et al.* [45] show that when system 1 is simulated by system 2 in this sense, an ACTL* formula satisfied by system 2 is also satisfied by system 1. ACTL* is the fragment of CTL* which uses only *universal* ($\forall$) path quantifiers.

Chapter 3

Modelling Embedded Systems

As was discussed in chapter 1, hard real-time systems often include a communications network linking computing nodes, require a software design that guarantees message transmission and response times, and the Controller Area Network is a popular choice in view of its deterministic collision avoidance strategy. This chapter therefore begins with a description of the Controller Area Network before moving on to a survey of embedded system modelling and model-checking techniques.

3.1 The Controller Area Network

Henceforth abbreviated CAN, this was originally devised by Bosch GmbH for use in automotive control applications [60, 74] where there was identified a need for high reliability in a noisy electromagnetic environment with data speeds of up to 1 Mbit/s and distances of up to a few tens of metres. Since then, CAN has become widely used in many hard-real-time systems, in medical and industrial applications including robotics.

Briefly, a CAN is a broadcast carrier-sense, multiple-access (CSMA-CA) network with a deterministic collision-avoidance strategy for medium access arbitration. It is a multi-master network: any node on the network may send a message. The message does not carry a destination address but is *broadcast* and may be received by any node interested in receiving it, in a sense which

will be explained further below.

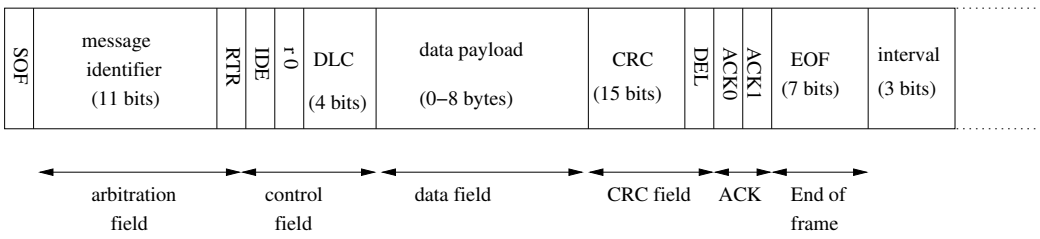


Figure 3.1: A CAN data frame

Messages passing over the network take the form of CAN data *frames* with the structure depicted in figure 3.1. From the point of view of communicating applications, the important elements are the 11-bit or 29-bit *message identifier* and the data pay-load of up to 8 bytes. We shall confine our attention to 11-bit identifiers in the rest of this section. The remaining elements are control bits and a cyclic redundancy check field used by lower level software to ensure a frame is correctly read from the network. The bits are physically encoded in NRZ.

The message identifier is not a unique sequence number but in fact plays two roles.

First, it expresses a message priority. Thought of as an integer, the lower its value, the higher the message priority. From this follows the deterministic procedure for arbitrating access to the network medium. A node may start to transmit a frame onto the network at any time. Nodes that wish to transmit monitor the medium while sending successive bits of their message identifier. As long as the competing nodes all send the same bit they remain in contention but as soon as a node sends a 1 bit while another node sends a 0 bit, the sender(s) of the 0 outbids the sender(s) of the 1 and the latter drops out of contention. One may speak of the 0 bit as "dominant" and of the 1 bit as "recessive". This bidding process repeats for all 11 bits of the identifier and provided two two nodes attempt to send a message with the same identifier at the same time there is a unique winner of the arbitration process – the message with the identifier with the smallest integer value[1].

[1]This implies that the message identifier is a *big-endian* integer, as the most significant

This scenario depends on clocks at the different nodes remaining in synchronisation with one another and also on the ability of a node writing an arbitration bit to sense when a "dominant" (0) bit is already on the bus and for the dominant bit to "win". These are provided in a straightforward way by the electronics on the CAN bus although this will not be described in detail here.

The second role performed by the message identifier is to announce a particular "type" of message. CAN messages are not addressed to a particular recipient – they are all *broadcast*; but a potential receiving node may choose to receive a frame of a particular type. A receiving node acts to receive a messages with a particular identifier value.

In a node a *CAN controller* senses the bus at all times and (typically) interrupts the host processor when a frame is sensed with an identifier identifying it as a frame which the node "wishes" to receive. Interrupt serving then consists of buffering the data of the frame.

In chapter 4, especially §4.2 and its sequel we shall adopt a rather abstract view of CAN frames/messages and their sending and receipt. The network will be viewed as divided in separate CAN *channels*. A message in a channel will be viewed as a pair consisting of an identifier and a data value, corresponding to the 11-bit identifier and 0-8 byte data value of figure 3.1. In §4.2 it is explained how a message-in-transit has an early *pre-acceptance* phase corresponding to medium access arbitration, a *post-acceptance* phase during which the message is "in" the medium and available to receivers, and an *acceptance* point which is the boundary between these. The network channel will also have a *status* recording whether it is free or, if it is handling a message, the progress of the message through its phases.

3.2 Broadcasting Embedded Systems

Steven Bradley, David Kendall, William Henderson and Adrian Robson have been particularly interested in modelling software systems distributed across

bit is presented first.

embedded hardware and communicating by broadcast – specifically, across controller area networks (CAN), initially developing a timed process algebra AORTA [36, 33, 32, 31, 37, 35, 34] and a response-time analysis tool $\mathcal{X}$RTA [66, 38, 65].

More recent work has led to the modelling language CANdle and its underpinning process algebra bCANdle [79, 80, 82, 81, 83, 84, 85]. This will be reviewed in some detail as it provides the point of departure for some of my work. [79] contains the most detail; the other works cited are papers summarising various aspects of this work.

3.2.1 CANdle

This is a high-level language for programming or modelling distributed real-time systems whose components communicate with CAN. A CANdle program consists of a number of *modules*: for instance ([79], p158):

```
module FlowRegulator is
  behaviour
    Flow[Msecs(10)/PERIOD] | Valve[y/x]
end module
```

This defines a module in terms of *instances* of two other modules running in parallel ([79], p147, p155).

```
module Flow is
  const
    PERIOD : duration
  type
    flow_rdng
  procedure
    ReadSensor(out flow_rdng)
  channel
    k : (flow.flow_rdng)
  var
    x : flow_rdng
  behaviour
    every PERIOD do
```

```
module Valve is
  type
    flow_rdng
  procedure
    AdjustValve(flow_rdng)
  channel
    k : (flow.flow_rdng)
  var
    x : flow_rdng
  behaviour
    loop do
      rcv(k, flow.x);
      AdjustValve(x)
```

```
      ReadSensor(x);                    end loop
      snd(k,flow.x)                  end module
    end every
end module
```

These module definitions show the the standard headings required by the language. The data types declared under heading `type` are defined separately – [79] provides a separate *Simple Data Modelling Language* (SDML) for this. The heading `channel` in each case declares a CAN channel with a message template consisting of *identifier* `flow` and *data payload* of type `flow_rdng`.

The `behaviour` sections exemplify two iteration constructs and the combination of actions (primitives, `snd`, `rcv` or procedure calls) in *sequence* (`;`). Actions can also be combined in *parallel* (`|`), as in the previous example and there are also constructs for *conditional action*, *non-deterministic choice* and trapping and handling of run-time exceptions.

The details of the procedures declared under `procedure` are abstracted out, although bounds on their execution time are needed for model-checking.

3.2.2 bCANdle

Chapter 6 of [79] goes on to describe how from a CANdle program is algorithmically generated a bCANdle system. bCANdle is a low-level process algebra used to provide a formal representation of these systems for model checking purposes.

A bCANdle system ([79], chapter 3; [81]) is made of three components: a *process term*, a *network model*, and a *data model*.

A data model over a given set of *variables*, a set of operation symbols and a set of predicate symbols consists of:

- a typing function assigning a set of values to each variable;
- a *valuation* - a mapping of variables to values;
- an assignment to each operation symbol of an operation on valuations of appropriate “arity”; and

- an assignment to each predicate symbol of a set of valuations.

If D is a data model and γ a predicate symbol, $D \models \gamma$ means that the valuation of D lies in the set which interprets γ. $D[x := v]$ is a data model like D except for assigning value v to variable x. If ω is an operation symbol, the relation $D \rightarrow_d^\omega D'$ means D' is the model like D except its valuation is the result of applying (the interpretation of) ω of D's valuation. One can thus see a data model *changing state*.

A network model consists of a set of broadcast *channels*. A channel consists of:

- a set of possible *messages*, each an *identifier* together with a data *value*
- a strict total ordering between messages (denoting priority);
- an assignment of upper and lower time bounds on the *pre-acceptance phase* and the *post-acceptance phase* of transmission of each message;
- a queue of waiting messages;
- a *state*: one of four: (1) free; or (2) in pre-acceptance phase of transmission of message m, with bounds l, u on time to completion; or (3) at acceptance point of transmission of message m; or (4) in post-acceptance phase of transmission of m, with bounds l, u on time to completion;

Rules are given for *changes of state* of the network, generating a timed transition system with network models at the nodes. The static parts of these network models are all the same; the dynamic parts - the queue and the state of the channel(s) - change as time passes or discrete actions occur. See [81], figure 3, or [79] §3.4.2.

The process terms are defined as follows. The style is that of Robin Milner's process algebra [98].

- $k!i.x$ and $k?i.x$ are process terms (transmission/reception of message $i.x$ through channel k);
- Where ω is an operation symbol and $t_1, t_2 \in \mathbb{R} \cup \{\infty\}$, $[\omega : t_1, t_2]$ is a process term (performance of an operation within given time bounds);

- Where γ is a predicate symbol and P a process term, $\gamma \rightarrow P$ is a process term (evaluate the guard and do P if satisfied);
- Where P, Q are process terms, $P; Q$, $P + Q$, $P[> Q$, $P|Q$ are process terms (do in sequence, non-deterministic choice, interrupt, do in parallel);
- A *process variable* X is a process term;
- $recX.P$ is a process term (recursion).

Milner[98] develops a theory of equivalence (bisimilarity) of process terms and shows how how the recursion construct represents iterative (looping) processes, and how any process term may generate an automaton. The locations are process terms, the generating term is the initial location, and a transition $a : P \rightarrow Q$, say, whenever P is able, in some sense, to perform action a and then must behave like Q, formalising the idea of a process algebra as a "list of things to do".

An analogous notion of bisimilarity equivalence is developed for bCANdle, based on rules for transition of a bCANdle system (P, N, D) into another, (P', N', D') whenever passage of time or some action has caused P to evolve in a fashion analogous to that described in the previous paragraph, or the network or data to change state. For instance, the process term $P = k!i.x; P'$ might evolve into P' and N to an N' that is like N but with message $i.x$ added to the queue, while $D = D'$. Another example: $N = N'$, the state of N is at acceptance point of $i.x$, $P = k?i.x; P'$, and D' is like D except the valuation has the value of variable x updated with the data just read.

The rules are stated in full in [79], §3.6, [81] figures 5 - 9 and explained in more detail in chapter 4.

A *clock* is added to each network channel ([79], chapter 4) to make a *clocked network*, $\hat{N}$, and to define from a process term P a *clocked process term* $\hat{P}$ by adding clocks to operations and building up using the same grammar as before. A *clocked bCANdle system* ([79]§4.2.4) is then a clocked process term, a clocked network and a data model. The automaton construction can then be strengthened to a *timed* automaton $\mathcal{G}(\hat{P}, \hat{N}, D)$ which

represents the possible behaviours of of the clocked bCANdle system starting from initial state $(\hat{P}, \hat{N}, D)$. Guards on transitions, for instance, ensure that time bounds on operations and on network behaviour are respected. The formal rules for generating the transitions of this timed automaton are given in [79], §4.3 and it is proved that the timed transition system which gives the operational semantics for it is strongly equivalent to that given by the bCANdle semantics previously defined.

We now have a formal translation of bCANdle systems into timed automata and the tools for analysing, checking timed automata can in theory be applied.

This work goes on to develop more algorithmically efficient ways to translate a bCANdle system, and indeed, to develop software to verify properties of a bCANdle system *on the fly* – without generating the equivalent timed automaton in its entirety. In my own studies, however, I have particularly been interested in the passage from bCANdle system to timed automaton and, using the detail just described, will describe my work with this in chapter 5.

3.3 Other Work on Model Checking Timed Automata

In this section a selection of papers are mentioned briefly. Much of this work is effectively summarised in the texts of Bérard *et al.* [26], Clarke *et al.* [45], Clark and Schlingloff [46].

3.3.1 Timed Automata and State Spaces

R Alur's and D L Dill's *A Theory of Timed Automata* [7] and *Automata-Theoretic Verification of Real-Time Systems* [8] describe some early work on getting time into finite automata. By comparison with §2.2, their automata include in the definition a subset $F \subseteq L$ of *acceptance states*, as in the "traditional" theory [73] where the actions form an *alphabet*, a word over the alphabet is *accepted* by an automaton if a run finishes at a location $\in F$

after the sequence of actions in the word. A classic theorem is that words accepted by finite automata over an alphabet A are precisely the regular expressions over A. Alur and Dill build time into this theory. They define a *timed word* as a sequence of letters (actions) with a matching sequence of time stamps and develop a notion of timed automaton similar to ours - a transition corresponding to a timed word can he taken if its time stamp satisfies a guard. Büchi acceptance is defined for timed runs: an ω-word (an infinite sequence of letters) is accepted if a location in F is visited infinitely often along the run. They also consider more general *Muller* acceptance, where a family of sets of locations is given and an ω-word is accepted if the set of locations visited infinitely often is in the family. Büchi acceptance is a special case of this.

This leads to a theory of *timed regular languages*, a natural generalisation of the regular languages of classical automata theory. They develop a PSPACE algorithm for checking the emptiness of the language accepted by a timed automaton and discuss other decision problems associated with their constructions. They show that timed Muller Automata and timed Büchi automata are equally expressive and properly include the language accepted by *deterministic* timed Muller Automata, which properly includes the language accepted by *deterministic* timed Büchi Automata.

Finally they apply their timed automaton construction of verifying safety and liveness properties of a real-time systems.

[8] discusses briefly the use of clock zones for efficient representation of the state space.

Alur, Courcoubetis, Halbwachs, Dill and Wong-Toi in [2] expound region equivalence as a way obtaining a finite state space, and develop an algorithm for constructing a *minimal* region graph for a timed automaton, and an algorithm for model-checking formulae of TCTL (see next section). Alur, Courcoubetis and Henzinger [4] develop the region graph construction further in order to obtain a solution to the *duration-bounded* reachability problem: where the timed automaton has a *duration measure* to decide whether there is a run satisfying a given upper bound on it.

Alur's paper *Timed Automata* [15] has developed the earlier theory into

essentially the same timed automaton formalism as that set out above in §§2.2, 2.2.3. In the course of a discussion of reachability analysis he develops the notion of region equivalence as described in §2.3 and shows that in a timed automaton of n locations and k clocks, in which every clock constraint constant is $\leq c$, the reachability problem can be solved in time $n.2^{O(klog(kc))}$. In considering implementation, he discusses the symbolic state space representation based on clock zones (§2.3) and their efficient representation using *difference-bound matrices*. Alur discusses again the *timed language* theory outlined in the previous paper and reviews a couple of tools, including UPPAAL.

J Bengtsson, B Jonsson, J Lilius and W Yi, [22] explore partial-order reduction of state-space search in the case of a parallel composite of timed automata, using a construct they call a *network of timed automata* instead of parallel composition as above. This allows them to define a *local time semantics*, a variation of the usual semantics of a parallel composition (see §2.2.3) according to which the components of a composite run independently; their clocks are not presumed to be synchronised *except* when a synchronizing transition is taken, when clocks are *resynchronised*. They develop from this a version of clock zone based symbolic states and a reachability algorithm with partial order reduction based on their local time semantics.

K G Larsen, F Larsson, P Pettersson and W Yi [88], [91] present timed automata as used in UPPAAL [50][89] (essentially as defined here) but with *networks of timed automata* as in the previous paragraph, symbolic semantics and difference bound matrices. They then obtain a number of results allowing non-minimal-weight edges to be dropped from the region graph without affecting the semantics. They develop an $O(n^3)$ algorithm for implementing this and also a method for reducing the number of symbolic states in the *global* PASSED list (see §2.3), and present a number of experimental results using UPPAAL.

The UPPAAL specification language is a subset of CTL that permits no nesting of $\forall\Box$, $\exists\Box$, $\forall\Diamond$, $\exists\Diamond$ modalities apart from that implied by its "leads to" construct, equivalent to $\forall\Box(\varphi \Rightarrow \forall\Diamond\psi)$. This apparent limitation of UPPAAL is rescued by the use of *test automata*, which allow specifications

to be checked which cannot be expressed directly in the property language. L Aceto, P Bouyer, A Burgueno and K G Larsen [1] describe the method of test automata. A test automaton is essentially a timed automaton in which some locations are deemed "reject states". A test automaton T expressing (the negation of) some specification is combined in parallel with an UPPAAL system which "passes the test" T if no timed state of the combined system which includes a reject location of T. This appears to be a way of using UPPAAL that goes beyond simple checking of properties expressed in its property (or specification) language. Aceto *et al.* go on to develop a powerful temporal logic $L_{\forall S}$ which augments a timed-automaton specification with extra boolean atomic formulae and clocks and expresses properties which can be reduced to reachability in this sense.

K G Larsen, C Weise, W Yi and J Pearson [90] develop *clock difference diagrams*, data structures inspired by binary decision diagrams ([45], ch 5) to provide an efficient representation of polyhedra such as clock zones.

G Behrmann, T Hune and F Vaandrager [20] explore running UPPAAL on parallel processors, distributing model-checking across processing nodes. This results in a search that is globally neither breadth-first (queue-organised search list) nor depth-first (stack-organised) but which approximates depth-first as the number of nodes increases. In fact, breadth-first is more optimal (and produces the shortest path to the pathological state in the case of a property failing); a prioritised queue of states to search is used to make distributed breadth-first search order closer to breadth-first search order. The speed-up in model-checking found in experiments was generally linear in the number of processing nodes, sometimes better.

E Fersman, L Mokrushin, P Pettersson and W Yi [54] consider the *schedulability problem* given a fixed-priority scheduling strategy, modelling a system of concurrent real-time processes as as a timed automaton with a set of tasks running at each location - a method of modelling different from the approaches considered so far. They show the problem of deciding whether a given task set encoded as a timed automaton in this way is schedulable can be solved by reachability analysis if two clocks are added to the automaton.

3.3.2 Timed Temporal Logics

Subsection 2.1 briefly reviewed some *temporal logics* used to express formally properties of real-time systems. There are many more – I feel I have barely scratched the surface and this is an area I might well return to in future work.

Rajeev Alur, Costas Courcoubetis and David Dill [17] extend computation tree logic, CTL, to a *timed* logic, TCTL. This logic has the ability to suffix the temporal operators $\Box, \Diamond, \mathcal{U}$ with a time bound $\sim c$, where $\sim$ is one of $<, \leq, =, \geq, >$ and $c \in \mathbb{R}$. Thus, for instance, $\exists\Diamond_{\leq 5}\varphi$ says: on some path, φ will be true in (up to) 5 time units; $\forall(\varphi\mathcal{U}_{<20}\psi)$ says: on all paths, ψ will be true less than 20 time units from now and in the mean time, φ is true. Their time line is continuous, not discrete, and their semantics replaces the discrete computation tree (timed transition graph) where each state has a discrete set of "next" states, with a continuous path through each state parametrised by time. They develop an algorithm based on clock zones and region graphs for checking TCTL formulae, and argue that having a dense rather than discrete time domain does not significantly increase the complexity of the model-checking problem.

T A Henzinger, Z Manna and A Pnueli [68] construct a similar timed extension to linear time logic (LTL or PTL) and show that for a large class of systems, checking a property over the continuous time domain can be reduced to a check over a discrete domain of a possibly modified property.

R Alur, K Etessami, S La Torre and D Peled [9] develop the logic of parametrised temporal operators further into a logic they call *parametric temporal logic*, PLTL. They note that some parametrised temporal operators are *upward monotone* in the intended interpretation, for instance $\Diamond_{\leq m}\varphi \Rightarrow \Diamond_{\leq m+1}\varphi$ and $\Box_{>m}\varphi \Rightarrow \Box_{>m+1}\varphi$; while their duals are *downward monotone*: e.g. $\Box_{\leq m+1}\varphi \Rightarrow \Box_{\leq m}\varphi$, $\Diamond_{>m+1}\varphi \Rightarrow \Diamond_{>m}\varphi$. This motivates them to carry two disjoint sets of parametric variables, one for upward and one for downward monotone operators. They develop model-checking for PLTL and sublogics based on Kripke structures.

Alur and Henzinger [10], [11] further discuss bounded temporal operators

in Timed PTL – $\Box(\varphi \rightarrow \Diamond_{[0,3]}\psi)$, $\Box(\varphi \rightarrow \Diamond_{\leq 3}\psi)$ and so forth. They also introduce *freeze quantification* in which, for instance in $\Box x(\varphi \rightarrow \Diamond y(\psi \wedge y \leq x+3)$, x, y are time variables bound to the time where the subformula in the scope of the temporal operator is evaluated. This formula says "in every state with time x, if φ, then there is a later state with time y such that $\psi \wedge y \leq x+3$). They compare these two logical constructs with a third syntax, quantification over explicit clock variables: for instance compare the previous formula with: $\forall x \Box((\varphi \wedge T = x) \rightarrow \Diamond(\psi \wedge T \leq x+3)$. [11] develops a tableau-based decision procedure for TPTL.

T Henzinger, X Nicollin, J Sifakis and S Yovine [69], develop a semantics of real-time systems somewhat different from the automaton-based semantics we have been working with, and a version of TCTL using explicit clock variables rather than parametrised temporal operators, and also a timed extension of the μ-calculus. They do a lot of work comparing the expressiveness of the two logics, concluding that they are not comparable in general, but that TCTL requirements of a sufficiently large class of real-time systems can be expressed in timed μ-calculus.

Alur and Henzinger have more recently been exploring game theoretic approaches to real-time system modelling and model-checking. Their approach (and also that O Kupferman [13]) develops models as games (the formal structure is a natural extension the automaton) in which the system and the environment alternate moves. They introduce a variety of temporal logic more general than LTL or CTL which they call *alternating time temporal logic*, ATL, which features a "selective" path quantifier $\langle\langle A \rangle\rangle$, where A is a set of players of the game. A formula $\langle\langle A \rangle\rangle \Box \varphi$, say, is evaluated at a state q of the game by considering a computation path forward from q in which repeatedly a *protagonist* chooses a move for every $a \in A$, then a *antagonist* chooses a move for every $a \in A^c$, and the state is updated with these moves. The authors argue that while LTL and CTL are suitable specification languages for *closed* systems, ATL is more natural for specifying *open* systems. They develop and examine the complexity of model-checking algorithms for ATL.

Alur and Peled [14] describe *causal* or *partial order* semantics which con-

trasts with the *interleaving* or *total order* models which we have been working with. The latter presume some kind of order between any two concurrent events, which the former allows that some pairs of events may in principle be incomparable as to causal order. They show that satisfiability in a number of temporal logics based on partial order semantics is undecidable.

3.4 Compositionality

If we aim to translate a bCANdle system into a parallel composite of stopwatch or hybrid automata, another avenue of enquiry suggests itself: how might *compositional* reasoning apply?

3.4.1 Assume-Guarantee Reasoning

Clarke *et al.* in [45] and [46] provide a brief introduction to this methodology, introduced more fully in Jones [75], Misra & Chandy [99], and Pnueli [104].

The basic idea is that one is aiming to verify a property of a composite system by inferring it from properties of components of the system. For instance, to verify that a communications protocol ensures that a message always gets from its source to its destination, one might separately verify (1) that a message always gets from its source onto the network, (2) that a message propagates through the network, and (3) that a message on the network is received by a "destination" process.

The assume-guarantee approach focuses on a number of *processes*, the *assumptions* made by each process about its environment, and the propositions *guaranteed* to be satisfied when the assumptions are met. The processes are expressed as formal models – process algebras, automata or other transition systems ([45] uses Kripke structures); the assumptions and guarantees are also expressed formally in some temporal logic such as CTL or its "universal" fragment, ACTL.

In the notation of Pnueli, $\langle\varphi\rangle\mathcal{M}\langle\psi\rangle$ is a *formula* intended to mean "whenever $\mathcal{M}$ is part of a system which satisfies assumption φ, the system guarantees that ψ is also satisfied".

A simple correctness proof is, for example,

$$\frac{\begin{array}{c}\langle\varphi\rangle\mathcal{M}'\langle\psi\rangle \\ \langle\mathtt{true}\rangle\mathcal{M}\langle\varphi\rangle\end{array}}{\langle\mathtt{true}\rangle\mathcal{M}||\mathcal{M}'\langle\psi\rangle}$$

A typical assume-guarantee proof, however, will involve system components $\mathcal{M}$, $\mathcal{M}'$ whose behaviour is mutually interdependent, and aim to specify assumptions to be satisfied by $\mathcal{M}'$ in order to guarantee the correctness of $\mathcal{M}$ and at the same time, specify assumptions to be satisfied by $\mathcal{M}$ in order to guarantee the correctness of $\mathcal{M}'$. To show that an appropriate combination of assumed and guaranteed properties of $\mathcal{M}$, $\mathcal{M}'$ guarantees the correctness of $\mathcal{M}||\mathcal{M}'$ might involve an inference of the form (see e.g. [46] p1767)

$$\frac{\begin{array}{c}\langle\varphi_1\rangle\mathcal{M}_1\langle\psi_1\rangle \\ \langle\varphi_2\rangle\mathcal{M}_2\langle\psi_2\rangle \\ \xi_1 \wedge \psi_1 \vdash \varphi_2 \\ \xi_1 \wedge \psi_2 \vdash \varphi_1 \\ \psi_1 \wedge \psi_2 \vdash \xi_2\end{array}}{\langle\xi_1\rangle\mathcal{M}_1||\mathcal{M}_2\langle\xi_2\rangle}$$

This sort of inference can be seen to be potentially circular if used carelessly [46]. Pnueli [104] discusses this and suggests indexing formulae with a well-founded set and allowing a formula to be deduced only from formulae with "lower" indices.

[46] mentions further discussion of this approach by Stephan Merz [97], B Josko [76], Long [93], Grumberg and Long [63].

Clark *et al.* [45] show how inferences such as these can be justified logically. They focus on systems with properties expressed in ACTL, the "universal" fragment of CTL which has a semantics over "fair" Kripke structures. The latter are Kripke structures with an additional family F of sets of states to define fairness: a run is fair iff at least one set in F is visited infinitely often by it. The parallel composition construction extends to fair structures in a natural way. They define $\mathcal{M} \models_F \varphi$ to mean φ is satisfied by the fair

Kripke structure $\mathcal{M}$, and extend the preorder $\preceq$ of 2.4.2 to version $\preceq_F$ for fair structures. This preorder has number of convenient properties:
– $\mathcal{M} \preceq_F \mathcal{M}' \models \varphi$ implies $\mathcal{M} \models \varphi$
– $\mathcal{M} \preceq_F \mathcal{M}'$ implies $\mathcal{M}||\mathcal{M}'' \preceq_F \mathcal{M}'||\mathcal{M}''$
– $\mathcal{M} \preceq_F \mathcal{M}||\mathcal{M}$ and $\mathcal{M}||\mathcal{M}' \preceq_F \mathcal{M}$
– Parallel composition is associative and commutative up to isomorphism.
– For each $\varphi \in ACTL$ there is a *tableau* or "canonical" model $\mathcal{T}_\varphi$ such that $\mathcal{M} \models \varphi$ iff $\mathcal{M} \preceq_F \mathcal{T}_\varphi$.

From these it is straightforward to render the Pnueli-style inferences illustrated above into ordinary logical inferences from these properties.

3.4.2 Assume-Guarantee Reasoning and Reactive Modules

This technique for deducing properties of a composite system from properties of its components is described in T A Henzinger, S Qadeer and S K Rajamani [70] in terms of a relation $P \preceq Q$ meaning *trace containment* – all traces or runs of P are (possible) traces of Q. They describe this relation in terms of *reactive modules*, described by R Alur and Henzinger in [12], although the technique can just as easily be thought of in terms of hybrid automata. It is useful to think of the relation as "P refines Q" or "P implements Q": Q is a specification, P a proposed implementation.

A *reactive module* P carries a set of *variables* X_P, partitioned into three subsets: *private*, *interface* and *external* variables. *Observable* variables are interface or external and are read by the module; *controlled* variables are interface or private and are set by the module. The interface variables are shared by several modules comprising a system.

Execution proceeds in *rounds*: the modules making up a system execute in round-robin fashion. Within a round, the *latched* values of the variables – values inherited from the previous round – are denoted $x, y, ...$; the *updated* values (set in the present round, usually by another module in the round-robin) by primes, $x', y', ...$

A module's procedures consist of *atoms*: every controlled variable is con-

trolled by an atom, which sets its values. An atom is defined by a sequence of guarded variable updates. Two simple examples from [12] should give the idea:

```
module Not
  external  in:B
  interface out:B
  atom out awaits in
    init update
      [] in' = 0 -> out' := 1
      [] in' = 1 -> out' := 0

module Latch
  external set,reset:B
  interface out:B
  private state:B
  atom out reads state
   [] true -> out' := state
  atom state awaits set, reset, out
   [] set' = 0 ^ reset' = 0 -> state' := out'
   [] set' = 1              -> state' := 1
   [] reset' = 1            -> state' := 0
```

Notice each atom corresponds to a controlled variable of the same name; `reads` means the atom works with a latched value of the variable; `awaits` means it works with an updated value. Clearly reactive modules are modellable at a lower level by hybrid automata. [12] describes an operational semantics in terms of *states* (valuations of the variables) and a successor relation between states, leading to a *transition graph*. A *trace* is the restriction to observable variables of a *trajectory* in the graph.

In terms of reactive modules, Q is *refinable by* P iff every interface variable of Q is an interface variable of P, and every external variable of Q is observable in P; $P \preceq Q$ iff these conditions are satisfied and every trace of P, restricted to Q, is a trace of Q. Q is *projection refinable* by P iff in addition, Q has no private variables.

These authors' assume-guarantee technique has the following outline. We would like, say, to show $P_1 \parallel P_2 \preceq Q$, but the state space of $P_1 \parallel P_2$ is too

large. Naïvely, we might show $P_1 \preceq Q$, $P_2 \preceq Q$ and draw our conclusion; but in practice these are strong premisses: normally $P_1 \preceq Q$ only in presence of suitable constraining assumptions about P_2. We construct A_2, an *abstraction* of P_2 encapsulating these constraints, and similarly A_1, an abstract description of P_1, and show $P_1 \parallel A_2 \preceq A_1 \parallel Q$ and $A_1 \parallel P_2 \preceq Q \parallel A_2$. An *assume-guarantee theorem* allows us to infer $P_1 \parallel P_2 \preceq Q$ from these.

To show $P' = P_1 \parallel A_2 \preceq A_1 \parallel Q = Q'$ is still PSPACE-hard in the size of the state space of Q'; however if the variables of Q' are all present in P' (Q' is projection refinable by P') then the complexity of the check is linear in the sizes of the state spaces of P', Q'. To contrive this, Henzinger *et al* [70] introduce a *witness* module to make explicit the way the hidden (private) variables of Q' depend on the state of P'. Then ([70] proposition 2), Q' is projection-refinable by $P' \parallel W$ and $P' \parallel W \preceq Q'$ implies $P' \preceq Q'$.

The human creativity required is to construct suitable abstraction and witness modules.

The *generalised* assume-guarantee rule ([70] proposition 3) runs as follows. Two reactive modules are said to be *compatible* is (1) their respective sets of controlled variables are disjoint, and (2) they do not jointly contain any cyclic *await dependencies*. An await dependency of a variable on one or more others is a dependency of an initial (or new) value of it on the initial (or new) values of other variables *in the same round*.

Now, say $P = P_1 \parallel ... \parallel P_n$ and say $Q = Q_1 \parallel ... \parallel Q_n$, refinable by P and for $i = 1...n$, Γ_i is a composition of *compatible* P_j, $Q_k (k \neq i)$. If $\Gamma_i \preceq Q_i$ for $i = 1...n$ then $P \preceq Q$. To use this, one decomposes Q into suitable components $Q_1 \parallel ... \parallel Q_n$ and for each Q_i, finds a suitable Γ_i, hopefully with a smaller state space than P. In general Γ_i is a composite of *essential modules*, P_j whose interface variables include all those of Q_i, and *constraining modules*, chosen from P_j, $Q_k (k \neq i)$ to make sure external variables of Q are all observable in Γ_i. The Q_ks are a better choice than the P_js – specification modules usually have smaller state spaces – but they may not provide enough constraints on the external variables of Γ_i and abstraction modules need to be added to provide these. One then appeals to ([70] proposition 4): If Q is refinable by P and $P \preceq Q \parallel A$, then $P \preceq Q$.

A detailed account of an application of these propositions is given in [70], sections 4 and 5.

3.4.3 Assume-Guarantee Reasoning and Timed I/O Automata

D Kaynar, N Lynch, F Vaandrager and R Segala [78, 77] develop an assume-guarantee reasoning technique rather similar to this, but formulated in terms of their *timed I/O automata*. This work has inspired an assume-guarantee theorem for UPPAAL timed automata described in chapter 7. Chapter 7 describes some of the detail of the constructions of Kaynar, Lynch *et al.*; here an overview of their overall approach is given, in some detail as they define many interesting concepts and constructions which inform the present work and possible future work following from it.

These authors start with a fairly standard definition of an (untimed) automaton, different only in that the set A of action labels is partitioned into *external* and *internal* actions: $A = E \cup H$. They develop this into a structure which they call a "timed automaton", which incorporates timed behaviour and is in fact an analogue not of our timed automaton, but rather of the timed transition system derived from it. This is defined below in §7.2.1 (page 131ff) where we have used the term "KLSV structure" to distinguish it from our timed automaton.

Interestingly, the KLSV structure focuses on states, i.e. valuations of variables, with no mention of control locations of an underlying automaton. A set X of variables is given, and the states are a designated subset Q of the valuations of X. A state q may change by a *discrete action* $a \in E \cup H$, $q \xrightarrow{a} q'$, or by evolution in time, along a *trajectory*, a function $\tau : J \to Q$ where J is an interval of the time line (usually $\mathbb{R}$) of the form $[0, b]$ or $[0, b)$ or $[0, \infty)$. Sets $\Theta \subseteq Q$ of *initial states*, $\mathcal{D} \subseteq Q \times (E \cup H) \times Q$ of possible discrete actions, and $\mathcal{T}$ of possible trajectories define the possible *executions* of the structure.

A trajectory τ is *from* q if $\tau(0) = q$. A *closed* trajectory is one with domain of the form $[0, b]$ and is *from* $\tau(0)$ *to* $\tau(b)$. An *execution fragment*

is an alternating sequence $\tau_0 a_1 \tau_1 a_2 \tau_2 a_3 ...$ of trajectories and discrete actions, starting with a trajectory and running to ∞ or else finishing with a trajectory; the trajectories apart from the last one of a finite fragment are closed and τ_k is from a_k to a_{k+1}. The execution fragment is *closed* if the sequence is finite and the final trajectory is closed. The execution fragment is *admissible* if, where τ_k has domain $[0, b_k]$, $\sum_{k=1}^{\infty} b_k = \infty$, and is *zeno* this sum converges to a finite value. An *execution* is a fragment that starts from a state in Θ. Kaynar, Lynch *et al.* define operations for obtain a *prefix* and a *suffix* of both a trajectory and an execution fragment, and for *concatenating* trajectories, execution fragments. The set $\mathcal{T}$ is required to be closed under taking prefixes, suffixes and concatenation.

Given an execution (fragment) $\tau_0 a_1 \tau_1 a_2 \tau_2 a_3 ...$ one can derive the underlying *trace* (fragment) by restricting valuations at each state to the empty set of variables, so that only the the amount of time elapsing figures, and restricting to external actions. If, for instance, action a_k drops out of the sequence, τ_k, τ_{k+1} are concatenated.

[77] explains that the timed transition system of an automaton of our type (which the authors call an "Alur-Dill automaton") can be modelled with these structures if X includes its clocks plus a variable `loc` to track the current *location* of the automaton. Θ consists just of the valuation that zeros all clocks and sets `loc` to the initial location of the automaton. $A = E \cup H$ is the action alphabet of the automaton; one can make $E =$ the automaton actions, $H = \emptyset$, or *vice versa*, or put synchronising actions in E and internal actions in H. The sets $\mathcal{D}$ and $\mathcal{T}$ are defined so as to respect the timed-transition semantics, and to ensure valuations of clocks have unit rate of change on a trajectory, and that valuations of `loc` are constant on a trajectory. See §7.2.2 for more detail.

Implementation relation. Kaynar, Lynch *et al.* call two structures $\mathbb{A}_i = (X_i, Q_i, \Theta_i, E_i, H_i, \mathcal{D}_i, \mathcal{T}_i)$, $i = 1, 2$ *comparable* if they have the same external actions, $E_1 = E_2$. In this case, it makes sense compare their external real-time behaviour as exhibited by traces. Accordingly, they say $\mathbb{A}_1$ *implements* $\mathbb{A}_2$, $\mathbb{A}_1 \preceq \mathbb{A}_2$ if the traces (from initial states) of $\mathbb{A}_1$ are included among those

of $\mathbb{A}_2$. They show that

- If every *closed* trace of $\mathbb{A}_1$ is in $\mathbb{A}_2$ and $\mathbb{A}_2$ has *finite non-determinism* (Θ_2 is finite and for every state x and trace fragment β from x, there only finitely many states reachable by execution fragments from x which restrict to β) then $\mathbb{A}_1 \preceq \mathbb{A}_2$.

- If every *admissible* trace of $\mathbb{A}_1$ is in $\mathbb{A}_2$ and $\mathbb{A}_1$ is *feasible* (from every state there is an admissible execution fragment) then every closed trace of $\mathbb{A}_1$ is a trace of $\mathbb{A}_2$.

- If every admissible trace of $\mathbb{A}_1$ is a trace of $\mathbb{A}_2$, $\mathbb{A}_1$ is feasible and $\mathbb{A}_2$ has finite non-determinism, then $\mathbb{A}_1 \preceq \mathbb{A}_2$.

Two sorts of *simulation* are defined in [77]. A *forward simulation* is analogous to ours of 2.4.2: a relation $R \subseteq Q_1 \times Q_2$ such that $\forall x_1 \in \Theta_1 \exists x_2 \in \Theta_2 : (x_1, x_2) \in R$, and such that whenever $(x_1, x_2) \in R$ and $x_1 \xrightarrow{\alpha_1} y_1$ is an execution fragment consisting of *either* a single discrete action (with a "point" trajectory at each end) *or* a single closed trajectory, then there is a closed execution fragment $x_2 \xrightarrow{\alpha_2} y_2$ with $(y_1, y_2) \in R$ and α_2 has the same trace as α_1. It follows as a theorem that whenever $(x_1, x_2) \in R$, any trace fragment from x_1 is a trace fragment from x_2; so that the existence of a forward simulation from $\mathbb{A}_1$ to $\mathbb{A}_2$ implies $\mathbb{A}_1 \preceq \mathbb{A}_2$. A composition of forward simulation relations is a forward simulation; a *refinement* is defined to be a forward simulation that is functional; a composition of refinements is a refinement. An *isomorphism* is a refinement whose inverse is also a refinement.

A *backward simulation* from $\mathbb{A}_1$ to $\mathbb{A}_2$ is defined by [77] to be a total ($\forall x_1 \in Q_1 \exists x_2 \in Q_2 : (x_1, x_2) \in R$) relation $R \subseteq Q_1 \times Q_2$ such that $\forall x_1 \in \Theta_1 \forall x_2 \in Q_2 : (x_1, x_2) \in R \Rightarrow x_2 \in \Theta_2$, and such that whenever $(y_1, y_2) \in R$ and $x_1 \xrightarrow{\alpha_1} y_1$ is an execution fragment consisting of *either* a single discrete action *or* a single closed trajectory, then there is a closed execution fragment $x_2 \xrightarrow{\alpha_2} y_2$ with $(x_1, x_2) \in R$ and α_2 has the same trace as α_1. Backward simulations compose as relations, and the existence of a backward simulation from $\mathbb{A}_1$ to $\mathbb{A}_2$ implies inclusion of *closed* traces.

The authors define a *history relation* from $\mathbb{A}_1$ to $\mathbb{A}_2$ as a forward simulation whose inverse is a refinement from $\mathbb{A}_2$ to $\mathbb{A}_1$ and they explore building an implementation relationship by *adding history variables* to $\mathbb{A}_1$ to make $\mathbb{A}_2$, so that $X_1 \subseteq X_2$, all states in Q_2 restrict to states in Q_1 and the relation $\{(y \upharpoonright X_1, y) | y \in Q_2\}$ is a history relation. Similarly they define a *prophecy relation* as a backward simulation whose inverse is a refinement and they explore building an implementation relationship by *adding prophecy variables* in the same way.

Composition. Kaynar, Lynch *et al.* define a parallel composition of *compatible* structures (disjoint sets of variables, each external action set disjoint from all other actions) which we describe in detail below in §7.2.3 (page 133ff) where we also show that it respects the parallel composition we have defined for our timed automata. They show that their construction is sound with regard to traces: a trace of $\mathbb{A}_1 \parallel \mathbb{A}_2$ projects to a trace of $\mathbb{A}_i$.

They show that

- If $\mathbb{A}_1$, $\mathbb{A}_2$ have the same external actions and are each compatible with $\mathbb{B}$, then $\mathbb{A}_1 \preceq \mathbb{A}_2$ implies $\mathbb{A}_1 \parallel \mathbb{B} \preceq \mathbb{A}_2 \parallel \mathbb{B}$.

- If $\mathbb{A}_1$, $\mathbb{A}_2$ have the same external actions, $\mathbb{B}_1$, $\mathbb{B}_2$ have the same external actions, and each $\mathbb{A}_i$ compatible with each $\mathbb{B}_j$, then $\mathbb{A}_1 \preceq \mathbb{A}_2$, $\mathbb{B}_1 \preceq \mathbb{B}_2$ imply $\mathbb{A}_1 \parallel \mathbb{B}_1 \preceq \mathbb{A}_2 \parallel \mathbb{B}_2$.

- If $\mathbb{A}_1$, $\mathbb{A}_2$ have the same external actions, $\mathbb{B}_1$, $\mathbb{B}_2$ have the same external actions, and each $\mathbb{A}_i$ compatible with each $\mathbb{B}_j$, then $\mathbb{A}_1 \parallel \mathbb{B}_2 \preceq \mathbb{A}_2 \parallel \mathbb{B}_2$, $\mathbb{B}_1 \preceq \mathbb{B}_2$ imply $\mathbb{A}_1 \parallel \mathbb{B}_1 \preceq \mathbb{A}_2 \parallel \mathbb{B}_2$.

Unfortunately, a desirable "assume-guarantee" result such as, under the same hypotheses as above, $\mathbb{A}_1 \parallel \mathbb{B}_2 \preceq \mathbb{A}_2 \parallel \mathbb{B}_2$, $\mathbb{A}_2 \parallel \mathbb{B}_1 \preceq \mathbb{A}_2 \parallel \mathbb{B}_2$ imply $\mathbb{A}_1 \parallel \mathbb{B}_1 \preceq \mathbb{A}_2 \parallel \mathbb{B}_2$ is not possible ([77] gives a counterexample, their pp 62-63) without further assumptions on the environments of the structures, which the authors address in constructing their "timed I/O automata".

Kaynar, Lynch *et al.* define other operations on their structures, such as

- *Hiding*, essentially moving a prescribed subset of E to H (the set of internal actions), which they show is monotonic with respect to their implementation relation $\preceq$.

- Adding lower and upper *time bounds* for a *task*, a designated subset of $E \cup H$ to an automaton, resulting in a stucture whose traces are included among those of the original.

- "Untiming" – quotienting states by a congruence relation to make a "plain old" automaton. This construction is reminiscent of symbolic state semantics and they employ it to prove a theorem slightly more general than that asserting the soundness of of symbolic state semantics for "Alur-Dill" automata.

Properties. In terms of KLSV structures, a *property* in [77] is set of execution fragments. In particular, a *safety* property is a set closed under prefixes and limits, corresponding to a requirement that needs to be maintained throughout an execution, and a *liveness* property P is such that *every* closed execution fragment can be concatenated with another to make a fragment $\in P$. The authors say $\mathbb{A}$ *satisfies* a safety property S if S contains every execution of $\mathbb{A}$, and a liveness property L if L contains every *maximal* (with respect to prefix) execution of $\mathbb{A}$. They show that any property which is both a safety and a liveness property contains *all* the execution fragments of $\mathbb{A}$ and that *any* property can be expressed as the intersection of a safety and a liveness property.

The authors develop a notion of *fairness* within this framework. If C is some set of actions of $\mathbb{A}$ ($C \subseteq E \cup H$), and x a state, say C is *enabled* in x if $\exists a \in C \exists y \in Q : x \xrightarrow{a} y$, otherwise say C is *disabled* in x. They define an execution fragment α to be *weakly fair* for C if α contains infinitely many events from C *OR* α has no suffix in all states of which C is enabled; if α has the stronger property that C is disabled in all states of some suffix, they say α is *strongly fair* for C. The set of strongly fair fragments for C is a liveness property, as is the set of weakly fair fragments.

They augment their structure: their *timed automata with properties* are

pairs $(\mathbb{A}, P)$ consisting of a KLSV structure and a property, the idea being to focus on executions of $\mathbb{A}$ that are in P. They develop a preorder $(\mathbb{A}_1, P_1) \preceq (\mathbb{A}_2, P_2)$ (the traces of $\mathbb{A}_1$ that are in P_1 are traces of comparable $\mathbb{A}_2$ in P_2) and prove that if P_1 is a liveness property, it follows from this that closed traces of $\mathbb{A}_1$ are traces of $\mathbb{A}_2$.

Given sets of actions $C_i \subseteq E_i \cup A_i,\ i = 1, 2$, they define a *fair forward simulation* from $\mathbb{A}_1$ to (comparable) $\mathbb{A}_2$ as a relation $R \subseteq Q_1 \times Q_2$ such that (1): $\forall x_1 \in \Theta_1 \exists x_2 \in \Theta_2 : (x_1, x_2) \in R$ and if C_1 is disabled in x_1, C_2 is disabled in x_2; and (2): Whenever $(x_1, x_2) \in R$ and $x_1 \xrightarrow{\alpha_1} y_1$ by either a single action surrounded by point trajectories or a single closed trajectory, there is a closed fragment $x_2 \xrightarrow{\alpha_2} y_2$ with the same trace as α_1, $(y_1, y_2) \in R$; and α_2 satisfies certain additional constraints relating enabledness/disabledness of C_2 in α_2 to enabledness/disabledness of C_1 in α_1: see [77]. It follows that the sets L_i of strongly fair executions of $\mathbb{A}_i$ are liveness properties and $(\mathbb{A}_1, L_1) \preceq (\mathbb{A}_2, L_2)$, and likewise when L_i are the sets of weakly fair executions of $\mathbb{A}_i$.

The authors similarly extend their parallel composition operation. Given properties P_i of $\mathbb{A}_i$, $i = 1, 2$, then define $P_1 \parallel P_2$ to be the set of fragments of $\mathbb{A}_1 \parallel \mathbb{A}_2$ which for each i restrict on variables and actions of $\mathbb{A}_i$ to fragments in P_i. Then they define $(\mathbb{A}_1, P_1) \parallel (\mathbb{A}_2, P_2)$ as $(\mathbb{A}_1 \parallel \mathbb{A}_2, P_1 \parallel P_2)$ and extend previous substitutivity theorems, such as

- When $\mathbb{A}_1$, $\mathbb{A}_2$ have the same external actions and are each compatible with $\mathbb{B}$, $(\mathbb{A}_1, P_1) \preceq (\mathbb{A}_2, P_2)$ implies $(\mathbb{A}_1, P_1) \parallel (\mathbb{B}, Q) \preceq (\mathbb{A}_2, P_2) \parallel (\mathbb{B}, Q)$

- When $\mathbb{A}_1$, $\mathbb{A}_2$ have the same external actions and $\mathbb{B}_1$, $\mathbb{B}_2$ have the same external actions and each $\mathbb{A}_i$ is compatible with each $\mathbb{B}_j$, $(\mathbb{A}_1, P_1) \preceq (\mathbb{A}_2, P_2)$ and $(\mathbb{B}_1, Q_1) \preceq (\mathbb{B}_2, Q_2)$ imply $(\mathbb{A}_1, P_1) \parallel (\mathbb{B}_1, Q_1) \preceq (\mathbb{A}_2, P_2) \parallel (\mathbb{B}_2, Q_2)$.

Timed I/O Automata Kaynar, Lynch *et al.* enhance their model by partitioning the external actions set E into *input actions* I and *output actions* O ($H \cup O$ comprise the *locally controlled* actions) and requiring that input actions are always enabled – $\forall x \in Q \forall a \in I \exists y :\ x \xrightarrow{a} y$ – and time passage

is always enabled – from every state there is a trajectory which is either defined on $[0.\infty)$ or closed, but some locally controlled action is enabled at the terminal state.

Executions, traces are defined as usual; a TIOA is *feasible* if the underlying timed automaton structure is, and *I/O-feasible* if its executions accommodate arbitrary input actions at arbitrary times. A TIOA $\mathbb{A}$ is *progressive* iff there are no zeno fragments of locally controlled actions, and *receptive* if it has a progressive *strategy* – an $\mathbb{A}'$ like $\mathbb{A}$ but with a subset of transitions ($\mathcal{D}' \subseteq \mathcal{D}$) and trajectories ($\mathcal{T}' \subseteq \mathcal{T}$). The authors show that a receptive TIOA is I/O-feasible.

The authors define two TIOAs to be *comparable* if the underlying structures are comparable and the O sets coincide and the I sets coincide. $\preceq$ is then the usual trace-inclusion, and is implied by the existence of a forward simulation.

Composition of compatible structures $\mathbb{A}_1, \mathbb{A}_2$ is defined as before, with $O = O_1 \cup O_2$ and $I = (I_1 \cup I_2) - O$ in the product. Substitutivity results are derived for TIOA essentially like the bulleted items on page 60 -

- If $\mathbb{A}_1$, $\mathbb{A}_2$ are comparable TIOA compatible with $\mathbb{B}$, then $\mathbb{A}_1 \preceq \mathbb{A}_2$ implies $\mathbb{A}_1 \parallel \mathbb{B} \preceq \mathbb{A}_2 \parallel \mathbb{B}$.

- If $\mathbb{A}_1$, $\mathbb{A}_2$ are comparable TIOA, $\mathbb{B}_1$, $\mathbb{B}_2$ are comparable TIOA, and each $\mathbb{A}_i$ compatible with each $\mathbb{B}_j$, then $\mathbb{A}_1 \preceq \mathbb{A}_2$, $\mathbb{B}_1 \preceq \mathbb{B}_2$ imply $\mathbb{A}_1 \parallel \mathbb{B}_1 \preceq \mathbb{A}_2 \parallel \mathbb{B}_2$.

- If $\mathbb{A}_1$, $\mathbb{A}_2$ are comparable, $\mathbb{B}_1$, $\mathbb{B}_2$ are comparable, and each $\mathbb{A}_i$ compatible with each $\mathbb{B}_j$, then $\mathbb{A}_1 \parallel \mathbb{B}_2 \preceq \mathbb{A}_2 \parallel \mathbb{B}_2$, $\mathbb{B}_1 \preceq \mathbb{B}_2$ imply $\mathbb{A}_1 \parallel \mathbb{B}_1 \preceq \mathbb{A}_2 \parallel \mathbb{B}_2$.

Most interestingly, there is now the assume-guarantee theorem that was not possible with the earlier KLSV structures: if $\mathbb{A}_1$, $\mathbb{A}_2$ are comparable, $\mathbb{B}_1$, $\mathbb{B}_2$ are comparable, and each $\mathbb{A}_i$ compatible with each $\mathbb{B}_j$, under certain additional assumptions (about traces in $\mathbb{A}_2$ $\mathbb{B}_2$), $\mathbb{A}_1 \parallel \mathbb{B}_2 \preceq \mathbb{A}_2 \parallel \mathbb{B}_2$, $\mathbb{A}_2 \parallel \mathbb{B}_1 \preceq \mathbb{A}_2 \parallel \mathbb{B}_2$ imply $\mathbb{A}_1 \parallel \mathbb{B}_1 \preceq \mathbb{A}_2 \parallel \mathbb{B}_2$. A careful statement of this theorem appears in §7.3.3.

There are limitations – for instance an example is given of a composition of I/O-feasible TIOAs which is not I/O-feasible; but a composition of progressive TIOAs is progressive and a composition of strategies is a strategy for the composite.

3.5 Abstraction and Abstraction Refinement

The sections above explore ways of reasoning about models of systems by way of a *simulation* or *trace containment* relation $\mathbb{A} \preceq \mathbb{A}'$ and rules for deducing a such a relation between *composite* models. Given such a relation, we can infer any universally quantified property (expressible, say, in ACTL or ACTL*) of $\mathbb{A}$ from that same property of $\mathbb{A}'$.

What is needed to complement this is a method for generating *abstract* models. Suppose it is wished to check a property φ, a formula in ACTL* of the form $\forall\Box\psi$, over a model $\mathbb{A}$. The present section explores an approach to generating an *abstraction* of $\mathbb{A}$ – a model $\mathbb{A}'$ such that $\mathbb{A} \preceq \mathbb{A}'$ – and in the event that φ fails in $\mathbb{A}'$, of determining whether the trace in $\mathbb{A}'$ providing the counterexample corresponds to a counterexample to φ in $\mathbb{A}$, or whether it is a *spurious* counterexample, an artefact of the abstraction that produced $\mathbb{A}'$. In the latter case, the approach derives a *refinement* of the abstraction tailored to eliminating the counterexample.

This approach, accordingly called *counterexample-guided abstraction refinement* (CEGAR) is developed by E Clarke, O Grumberg, S Jha, Y Lu and H Veith in [44].

These authors develop the method with reference to *Kripke structures* – see §§ 2.1.2, 2.3.1.

3.5.1 Programs and Existential Abstraction

Clark *et al.* derive their Kripke structures from "concurrent programs". A program consists of a set of variables $\{v_1, ..., v_n\}$, variable v_i having a domain D_i. The Cartesian product $S = D_1 \times ... \times D_n$ forms the state space of the Kripke structure.

Arithmetic *expressions* are formed of arithmetic operators, variables and constants from the D_i; *atomic formulae* from expressions and relation symbols ($<, \leq, =, \geq, >$ etc) and *predicates* from boolean combinations of atomic formulae. If $d = (d_1, ..., d_n) \in S$ and φ is a predicate, $d \models \varphi$ means φ holds when (for each i) d_i is substituted for v_i.

A "concurrent program" is formed by, for each i, a *transition block* describing the evolution of values of v_i:

$$
\begin{array}{l}
init(v_i) := I_i \\
next(v_i) := \text{case} \\
\qquad C_i^1 : A_i^1 \\
\qquad \cdots \\
\qquad C_i^m : A_i^m \\
\text{esac}
\end{array}
$$

where $I_i \subseteq D_i$, declaring the possible initial values of v_i, the C_i^k ($k = 1...m$) are predicates and the A_i^k expressions. The A_i^k describe changes of state *guarded* by C_i^k.

The derived Kripke structure (S, I, R, L) has initial state set $I = I_1 \times ... \times I_n$ and the transition relation $R \subseteq S \times S$ derived from the $next(v_i)$ clauses. Recall that L is a function which associates with each state a set of predicates "true there". In the present case, assuming in addition to the program a predicate φ to be checked, $L(d)$ ($d \in S$) is the subset of "atoms of the program" (atomic formulae which occur positively or negatively in the C_i^k or in φ) α such that $d \models \alpha$.

Clark *et al.* define an *existential abstraction* of such a program or Kripke model in terms of a surjective function on the set of states S. Equivalently, this can be defined as an equivalence relation $\equiv$ on S. The *abstraction* of $\mathbb{M} = (S, I, R, L)$ is then defined as a Kripke structure $\hat{\mathbb{M}}$ with state set $\hat{S} = S/\equiv = \{\hat{s} : s \in S\}$[2]. The initial states of the abstraction are $\hat{I} = I/\equiv$ and the abstract transition relation $\hat{R} = \{(\hat{s}, \hat{t}) \in \hat{S} \times \hat{S} : R(s, t)\}$. The abstract labelling function is

[2] $\hat{s} \triangleq \{s' \in S : s \equiv s'\}$, the $\equiv$-equivalence class determined by s.

$$\hat{L}(\hat{s}) = \bigcup_{s' \in \hat{s}} L(s')$$

It is assumed that the abstraction relation $\equiv$ is *appropriate for* φ in that related states agree as to the truth of all sub-formulae of φ. $\hat{L}$ labels abstract states *consistently*. This assumption guarantees that the abstraction produces no "false positives"; it *simulates* the base model – $\mathbb{M} \preceq \mathbb{M}'$ and $\mathbb{M}' \models \varphi \Rightarrow \mathbb{M} \models \varphi$.

3.5.2 Generating an Abstraction

For any set F of formulae, an equivalence relation between states is defined by $s \equiv_F t$ iff for every $\psi \in F$, $s \models \psi \Leftrightarrow t \models \psi$. Given a concurrent program P and specification to be checked, φ, Clarke *et al.* define an *initial* abstraction relation relative to the atoms of P, φ: $s \equiv_{init} t$ iff for every atom α of P, φ, $s \models \alpha \Leftrightarrow t \models \alpha$.

Clarke *et al.* ([44], §4.2) state that whereas there are many possible equivalence relations on $S = D_1 \times ... \times D_n$, hence many possible abstractions of the initial model, the "interesting" ones are generally made component-wise. The most extreme way of doing this is by constructing the equivalence from component equivalences $\equiv_i$ on D_i $i = 1...n$; $(d_1, ..., d_n) \equiv (e_1, ..., e_n)$ iff $\bigwedge_{i=1}^{n} d_i \equiv_i e_i$.

They explore a generalisation of this based on what they call *formula clusters* and *variable clusters*. They say that two formulae *interfere* if they have a variable in common. The formula clusters are the equivalence classes of the reflexive transitive closure of the interference relation between atoms. They then define $v_i \equiv v_j$ iff the variables appear in atomic formulae in the same formula cluster; the variable clusters are the equivalence classes of this relation.

Where $F_1, ..., F_m$ are the formula clusters and $V_1, ..., V_m$ the variable clusters so defined, let $S_k \triangleq \prod_{i:v_i \in V_k} D_i$. An equivalence is defined on S_k: for $d, e \in S_k$, $d \equiv_k e$ iff $\bigwedge_{\alpha \in F_k} (d \models \alpha \Leftrightarrow e \models \alpha)$. These equivalences, by construction, "glue together" to make an equivalence relation on the full state

space S.

Clarke *et al.* show ([44], §4.2.3, their theorem 4.8) that the abstraction induced by this equivalence relation is isomorphic to the abstraction defined by the relation $\equiv_{init}$ defined above, at the beginning of this subsection.

They also remark that it is in general computationally cheaper for a model-checking software tool to compute an *over-approximation* to the abstraction: $\tilde{\mathbb{M}} = (\tilde{S}, \tilde{I}, \tilde{R}, \tilde{L})$ where $\tilde{S} = \hat{S}$, $\tilde{L} = \hat{L}$, but $\tilde{I} \supseteq \hat{I}$, $\tilde{R} \supseteq \hat{R}$. This in general coarsens the abstraction: $\mathbb{M} \preceq \hat{\mathbb{M}} \preceq \tilde{\mathbb{M}}$ but this is not a problem and the next stage of their technique is iterated *refinement* of the abstraction.

3.5.3 Spurious Counterexamples

When $\mathbb{M} \preceq \hat{\mathbb{M}}$ and $\hat{\mathbb{M}} \models \varphi$ one can infer $\mathbb{M} \models \varphi$ assuming φ is universally quantified – e.g. an ACTL* formula. However, if $\hat{\mathbb{M}} \not\models \varphi$, the counterexample trace in $\hat{\mathbb{M}}$ may be *spurious*, not corresponding to a counterexample in $\mathbb{M}$. Clarke *et al.* show that such spuriousness arises as follows.

The abstract trace will have the form $\hat{s}_0, \hat{s}_1, ..., \hat{s}_N$, a finite sequence of equivalence classes on concrete states, where $(\forall i < N)(\hat{s}_i, \hat{s}_{i+1}) \in \hat{R}$. Now, $\hat{R}$ relates abstract states $\hat{s}, \hat{t}$ whenever they, as equivalence classes, have members $s' \in \hat{s}$, $t' \in \hat{t}$ related by R.

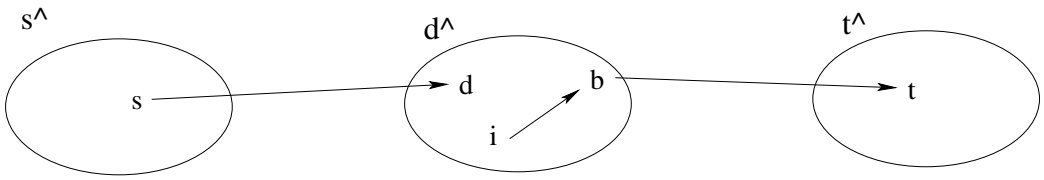


Figure 3.2: Part of a spurious counterexample

It is possible, therefore that the counterexample contains a sequence $\hat{s}, \hat{d}, \hat{t}$ like that depicted in figure 3.2. The ovals represent abstract states (equivalence classes of concrete states) and the arrows denote *related* concrete states: for instance, $s \rightarrow d$ indicates $(s, d) \in R$. State d has transition *to* it but no transitions *from* it: it is a *dead-end* state. The abstract trace $\hat{s} \rightarrow \hat{d} \rightarrow \hat{t}$ is possible because there exists a state $b \in \hat{d}$ (i.e. $b \equiv d$) with a transition to a

state $t \in \hat{t}$.

State b is a *bad* state. It raises the possibility of a transition sequence $\hat{s} \rightarrow \hat{d} \rightarrow \hat{t}$ corresponding to *no* transition sequence $s' \rightarrow d' \rightarrow t'$ between concrete states $s \in \hat{s}$, $d \in \hat{d}$, $t \in \hat{t}$. The trace is spurious *because* of the existence of a bad state in the same equivalence class as a dead-end state, spuriously connecting two execution sequences which would be disconnected in the concrete model.

State i is an *irrelevant* state – its existence and relatedness to other states is neither "good" nor "bad" for the counterexample.

Clark *et al.* argue ([44], §3) that such a spurious counterexample can be eliminated by *refining* the abstraction relation by splitting the equivalence class that contains the dead-end and the bad state, so that these states end up in *different* equivalence classes. In the method they develop this process is iterated until φ is verified, or until a counterexample is found that is not spurious, and can be pulled back into the concrete model.

The authors develop algorithms for identifying spurious counterexamples, implementable (using data representations such as OBDDs) in model-checking software. Their lemma 4.10 [44] shows that an abstract counterexample $(\hat{s}_1, ..., \hat{s}_n)$ corresponds to a concrete counterexample $(s_1, ..., s_n)$ [3] iff the sets $\Sigma_1, ..., \Sigma_n$ of concrete states defined recursively by $\Sigma_1 = \hat{s}_1 \cap I$, $\Sigma_i = \hat{s}_i \cap R(\Sigma_{i-1})$ are all non-empty[4]. Indeed, when $(\hat{s}_1, ..., \hat{s}_n)$ does correspond to an concrete path $(s_1, ..., s_n)$ which is a counterexample, $s_i \in \Sigma_i$ for $i = 1, ..., n$ while conversely if all the Σ_i are non-empty, a concrete counterexample can be constructed by starting with a state in Σ_n and working backwards inductively: from $s_i \in \Sigma_i = \hat{s}_i \cap R(\Sigma_{i-1})$ we infer an R-pre image $s_{i-1} \in \Sigma_{i-1} \subseteq \hat{s}_{i-1}$ and so build up $(s_1, ..., s_n)$.

Their algorithm `SplitPATH` ([44], §4.3.1) is based on this. Given a finite

[3] $s_1 \in I$; for $i = 2, ..., n$ $(s_{i-1}, s_i) \in R$; $s_i \in \hat{s}_i$, i.e. $\hat{s}_i$ is the equivalence class determined by s_i;

[4] $R(\Sigma) \triangleq \{s \in S : \exists t \in \Sigma : R(t, s)\}$, the image under R of Σ.

abstract path $(\hat{s}_1, ..., \hat{s}_n)$ in $\hat{\mathbb{M}}$,

$$
\begin{aligned}
&\Sigma := \hat{s}_1 \cap I \\
&i = 1 \\
&while(\Sigma \neq \emptyset \wedge i < n) \\
&\qquad i := i + 1 \\
&\qquad \Sigma_{prev} := \Sigma \\
&\qquad \Sigma := R(\Sigma) \cap \hat{s}_i \\
&if\ (\Sigma \neq \emptyset)\ output\ \text{“counterexample exists”} \\
&else\ output\ i, \Sigma_{prev}
\end{aligned}
$$

The algorithm computes whether there is a concrete path $(s'_1, ..., s'_n)$ in $\mathbb{M}$ which maps to the abstract path[5]: From their lemma 4.10 [44], Clarke *et al.* deduce that if the counterexample is spurious, `SplitPATH` terminates with the smallest i for which the set Σ is empty, and Σ_{prev} contains dead-end states, for in this case Σ is an empty Σ_i in the notation of the previous paragraph, and Σ_{prev} is the corresponding Σ_{i-1}.

Before seeing how these data are used, they consider the alternative possibility that the abstract counterexample is a loop: $(\hat{s}_1, ..., \hat{s}_k)(\hat{s}_{k+1}, ..., \hat{s}_n)^*$. Their analysis of this possibility shows ([44], §4.3.2) that the loop portion of it needs to be “unwound” at most a polynomial number of times and that the abstract counterexample pulls back to a concrete counterexample or is spurious according to whether the “unwound” finite trace does/is.

Their algorithm `SplitLOOP` does this. First the minimum of the sizes of the equivalence classes $\hat{s}_{k+1}, ..., \hat{s}_n$ is computed, then (this minimum + 1) unwindings of the abstract loop counterexample is computed and `SplitPATH` applied to the result. If this yields a concrete counterexample this result is returned. Otherwise, `SplitPATH` has returned an index number i and the dead-end set Σ_{prev}. Indices in the original loop counterexample corresponding to i, $i + 1$ are computed and returned along with Σ_{prev}.

Iterated abstraction refinement follows these steps.

[5] $(\forall i) s'_i \in \hat{s}_i$ and $(\forall i < n)(s'_i, s'_{i+1}) \in R$

3.5.4 Refining the Abstraction

Algorithm `SplitPATH` returns, in the case of a spurious counterexample, a set Σ_{prev} of dead-end (concrete) states reachable (compatibly with the abstract trace) from I. It also returns the index number i of a state $\hat{s}_i$ of the abstract trace, such that $\Sigma_{prev} \subseteq \hat{s}_{i-1}$ but $R(\Sigma_{prev}) \cap \hat{s}_i = \emptyset$. There *is*, however, an abstract transition $\hat{R} : \hat{s}_{i-1} \rightarrow \hat{s}_i$ (spurious!), so the set $\Sigma_B \triangleq \{s \in \hat{s}_{i-1} : \exists s' \in \hat{s}_i : (s, s') \in R\}$ is non-empty

This subset of $\hat{s}_{i-1}$ is a set of *bad* states and is disjoint from $\Sigma_D \triangleq \Sigma_{prev}$, a set of dead-end states. Defining $\Sigma_I \triangleq \hat{s}_{i-1} - \Sigma_D - \Sigma_B$ yields a partition $\hat{s}_{i-1} = \Sigma_D \sqcup \Sigma_B \sqcup \Sigma_I$.

The aim of the refinement step is to find the coarsest refinement of the abstraction relation $\equiv$ which *separates* Σ_D, Σ_B.

Recalling that the concrete state space S has the form $D_1 \times ... \times D_n$, the set of n-tuples of values of the variables, Clark *et al.* express the relation $\equiv$ as a "composite" of equivalence relations $\equiv_i$ on D_i (i.e., $(d_1, ..., d_n) \equiv (e_1, ..., e_n)$ iff $\bigwedge_{i=1}^{n} d_i \equiv_i e_i$) and write the equivalence class $\hat{s}_{i-1}$ to be split as a product $E_1 \times ... \times E_n$ of $\equiv_i$-equivalence classes. They then give a polynomial-time algorithm `PolyRefine` to construct a *refinement* $\equiv_i'$ of each $\equiv_i$:

$$
\begin{array}{l}
for\ (i := 1\ to\ m) \\
\quad \equiv_i' := \equiv_i \\
\quad for\ (a, b \in E_i) \\
\qquad if\ (proj(\Sigma_D, i, a) \neq proj(\Sigma_D, i, b)) \\
\qquad\quad \equiv_i' := \equiv_i' - \{(a, b)\}
\end{array}
$$

Here, $proj(\Sigma_D, i, a)$ denotes the set of $(n-1)$-tuples which yield an n-tuple in Σ_D when a is inserted in position i. Clarke *et al.* show ([44], lemma 4.19 and corollaries) that the relation $\equiv'$ made by composing the $\equiv_i'$ is an equivalence relation on states S, is a refinement of $\equiv$, and is the coarsest one which separates Σ_D and Σ_B. It does in fact separate Σ_D from $\Sigma_B \cup \Sigma_I$.

The refinement procedure for a loop counterexample is analogous. The refinement procedure is performed repeatedly until either a concrete coun-

terexample is found or the property φ is verified. Partitioning must terminate because every equivalence class contain at least one element.

The method described here was developed by Clark *et al.* for implementation in a software tool using symbolic states and OBDDs; but it will be interesting to investigate how aspects of the method may be used to refine our compositional models.

Chapter 4

bCANdle

This chapter gives, for convenient reference, the key definitions of bCANdle, [79], a framework for modelling broadcasting embedded systems as timed transition systems. The following two chapters will develop from this a semantically equivalent *compositional* modelling framework.

A bCANdle system comprises three components: a *data model*, a *communication model*, and a *process model*.

4.1 The Data Model

Fix V, a set of data values, and Var, a (finite) set of data variables.

- A *valuation* is a total mapping $Var \longrightarrow V$.
- An *operation* is a binary relation between valuations, total but not necessarily one-to-one. (Operations need not be deterministic.)
- A *predicate* is a set of valuations.

Given, in addition to Var, finite sets Ω of operation names and Γ of predicate names, a *data environment* over Var, Ω, Γ is a tuple $D = (type, op, pred, val)$ where

- *type* maps each variable name to a subset of V. If $x \in Var$, $type(x) \subseteq V$ is the "universe" of values of x.

- *op* maps each operation name to an operation. If $\omega \in \Omega$, $op(\omega) \subseteq V^{Var} \times V^{Var}$.

- *pred* maps each predicate name to a set of valuations. For $\gamma \in \Gamma$, $pred(\gamma) \subseteq V^{Var}$ is the "extension" of γ.

- $val : Var \longrightarrow V$ gives each $x \in Var$ its "current" value: $val(x) \in type(x)$.

If the data environment needs to be made explicit, one writes $D.type$, $D.op$, etc. For $x \in Var$, $D.x$ is an abbreviation for $D.val(x)$.

$D[x := v]$ denotes a data environment D' like D except $D'.x = v$; i.e. $D'.y = D.y$ for all $y \in Var - \{x\}$.

For $\omega \in \Omega$, the notation $D \xrightarrow{\omega}_d D'$ defines a relation describing how a data environment may evolve: it denotes the condition $D.type = D'.type \wedge D.op = D'.op \wedge D.pred = D'.pred \wedge (D.val, D'.val) \in D.op(\omega)$. In short, $D \xrightarrow{\omega}_d D'$ says that D' is "D after operation ω has run".

Ω contains a symbol ID which is always interpreted as the identity relation between valuations.

For $\gamma \in \Gamma$, $D \models \gamma$ abbreviates $D.val \in D.pred(\gamma)$.

Two data environments are *compatible* if they agree apart possibly from their *val* components.

4.2 The Network Model

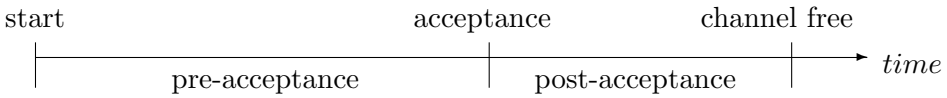
This section describes an abstract representation of a controller area network as described in §3.1.

A (CAN) *network* consists of a number of *broadcast channels* through which *messages* pass.

Assume given a finite set I of *message identifiers* and a finite set V of data values. The identifiers correspond to the arbitration bits of a physical CAN frame – they express message priority as far as medium access is concerned while, from the point of view of application software, they can be thought off as identifying message *type*. The data values correspond to the data payload

of the CAN frame. In the present abstract formulation, a message is defined formally as pair $(i, v), i \in I, v \in V$; this is also denoted $i.v$. I is presumed to be equipped with a strict total ordering $i \preccurlyeq i'$ defining *priority* ordering between messages. $m = (i, v) \preccurlyeq (i', v') = m'$ iff $i \preccurlyeq i'$.

A message in transit has an early *pre-acceptance* phase during which the sender is negotiating for access to the network medium, followed by a *post-acceptance* phase during which the message is "in" the medium and available to receivers. The *acceptance* point is the boundary between these phases.



The *transmission latency* is the elapsed time from the start of transmission until the channel is free again. A *transmission latency function* assigns to a message $m = i.v$ a vector of four real numbers $\delta(m) = (l, u, L, U)$, which are time bounds on the durations of the phases: $l \leq preacceptance \leq u$, $L \leq postacceptance \leq U$.

A *channel* delivers messages. Let M be a set of messages and δ a transmission latency function as above. A channel, at any time, has a *status*, denoted by one of the following symbols:

- $\downarrow$ - the channel is free;
- $\overset{t_1,t_2}{\rightsquigarrow} m$ - the channel is in pre-acceptance phase of m, with lower bound t_1 and upper bound t_2 on time to completion ($m \in M, 0 \leq t_1 \leq l(m), 0 \leq t_2 \leq u(m), t_2 - t_1 \leq u(m) - l(m)$ with equality when $t_1 > 0$);
- $\uparrow m$ - the channel at acceptance point of message m;
- $m \overset{t_1,t_2}{\rightsquigarrow}$ - channel is in post-acceptance phase of m, with lower bound t_1 and upper bound t_2 on time to completion ($m \in M, 0 \leq t_1 \leq L(m), 0 \leq t_2 \leq U(m), t_2 - t_1 \leq U(m) - L(m)$ with equality when $t_1 > 0$);

A *message queue* is a sequence of messages in *priority* order: $j < k$ implies $m_j \prec m_j$ and (when $j \neq k$) m_j, m_k have distinct identifier (type)

values. There is thus at most one message in a queue with a given identifier value. Since I is finite, it follows that a queue always has finite length. The notation $m : u$ denotes a queue which is u with message m at the "head" and u as the "tail".

A message may be inserted into a queue: if u is a message queue and $i.v$ a message, $u \leftarrow\!\!\!\rho\ i.v$ is the queue state which is the result of inserting $i.v$ into u in correct position (overwriting any message with identifier i already in u).

To summarise, a *channel* is a set of messages with a priority ordering, a latency function, a status as defined above, and a message queue. While the other components are fixed, the status s and queue u are dynamic, varying as the system evolves: so the channel may simply be denoted (s, u).

A *network* N is an indexed set of channels, $N_k : k \in K$. The notation $N[k := \eta]$ denotes an *updated* network N' such that $N'_k = \eta$ and N'_j for $j \neq k$.

Network *behaviour* can now be defined by means of a relation $\longrightarrow_n$. A network N (a set of channels each in particular state) may evolve into N' on passage of an interval of time $t \in \mathbb{R}$: this is denoted $N \xrightarrow{t}_n N'$. N may evolve into N' on the occurrence of a *discrete action* a: denoted $N \xrightarrow{a}_n N'$. The set A_n of discrete actions consists of the union of the following, where K indexes the channels of the network, I is the set of message identifiers, and V the set of data values.

- $\{\ k \rightsquigarrow i.v : k \in K, i \in I, v \in V\ \}$ - enter pre-acceptance phase of message $i.v$ on channel k;
- $\{\ k \uparrow i.v : k \in K, i \in I, v \in V\ \}$ - acceptance of message $i.v$ in channel k;
- $\{\ i.v \rightsquigarrow k : k \in K, i \in I, v \in V\ \}$ - enter post-acceptance phase of message $i.v$ on channel k;
- $\{\ k \downarrow : k \in K\ \}$ - channel k becomes free;

Changes of network state, $N \longrightarrow_n N'$, resulting from these discrete actions, are now defined precisely by the following *rules of inference.*

$$\frac{N_k = (\downarrow, m : u)}{N \xrightarrow{k \rightsquigarrow m}_n N[k := (\overset{l(m),u(m)}{\rightsquigarrow} m, u)]} \tag{N1}$$

If N_k is a free channel with a non-empty queue, the network may enter the pre-acceptance phase on channel k, de-queuing the message.

$$\frac{N_k = (\overset{0,t}{\rightsquigarrow} m, u)}{N \xrightarrow{k \uparrow m}_n N[k := (\uparrow m, u)]} \tag{N2}$$

When the lower bound on time to completion of the pre-acceptance phase of transmission of a message becomes 0, the network may *accept* it.

$$\frac{N_k = (\uparrow m, u)}{N \xrightarrow{m \rightsquigarrow k}_n N[k := (m \overset{L(m),U(m)}{\rightsquigarrow}, u)]} \tag{N3}$$

From the state of acceptance of message m, channel k may enter the post-acceptance phase, with bounds $L(m), U(m)$ on time to completion.

$$\frac{N_k = (m \overset{0,t}{\rightsquigarrow}, u)}{N \xrightarrow{k \downarrow}_n N[k := (\downarrow, u)]} \tag{N4}$$

When the lower bound on time to completion of the post-acceptance phase of transmission of a message becomes 0, the channel may become free.

These four rules specify the changes of state of a network effected by the four types of discrete action. It remains to to provide a rule for change of state due to *passage of time*.

To assist in formulating his rule, let $\texttt{tcp}(s, u)$ denote the *maximum time progress* allowed to a channel in state (s, u). $\texttt{tcp}(\downarrow, u) = \infty$ if queue u is empty, otherwise 0. For $t_0, t \in \mathbb{R}, \texttt{tcp}(\overset{t_0,t}{\rightsquigarrow} m, u) = \texttt{tcp}(m \overset{t_0,t}{\rightsquigarrow}, u) = t$. Lastly, $\texttt{tcp}(\uparrow m, u) = 0$. For a complete network state N, define $\texttt{tcp}(N) = min_k \texttt{tcp}(N_k)$.

The *effect* of time progress is to transform a channel state (s, u) to a state $(s, u) + t$ defined thus:

- $(\downarrow, u) + t = (\downarrow, u)$ if u is empty or if $t = 0$; otherwise it is *undefined*;
- $(\overset{t_1,t_2}{\rightsquigarrow} m, u) + t = (\overset{t'_1,t'_2}{\rightsquigarrow} m, u)$ if $t \leq t_2$; otherwise it is undefined;

$t'_1 = max(t_1 - t, 0)$ and $t'_2 = t_2 - t$;

- $(\uparrow m, u) + t = (\uparrow m, u)$ if $t = 0$, otherwise it is undefined;
- $(m \overset{t_1,t_2}{\rightsquigarrow}, u) + t = (m \overset{t'_1,t'_2}{\rightsquigarrow}, u)$ if $t \leq t_2$; otherwise it is undefined;
  $t'_1 = max(t_1 - t, 0)$ and $t'_2 = t_2 - t$.

Time progress for a network is just time progress for all its channels, where defined. For those $k \in K$ for which the right hand side is defined, $(N + t)_k = N_k + t$.

We can now formulate the passage-of-time rule:

$$\frac{\forall k \in K : 0 \leq t \leq \texttt{tcp}(N_k)}{N \xrightarrow{t}_n N + t} \tag{N5}$$

4.3 The Process Model

A bCANdle *process* is described by a simple process-algebraic language. Assume given a network model supplying a set K of channel identifiers and a set I of message identifiers; and also a data model which supplies sets Var of variables, Ω of operation names, Γ of predicates, as explained in the previous subsections. Assume also a countable set of *process variables*.

A *process term* is one of the following.

- $k!i.x$
  where $k \in K, i \in I, x \in Var$. This denotes a process which causes a message $i.val(x)$ to be queued instantly for sending on channel k, and then terminates.
- $k?i.x$
  where $k \in K, i \in I, x \in Var$. This denotes a process which waits for a message with identifier i to reach its acceptance point on channel k; then the data is associated with x and the process terminates.
- $[\omega : t_1, t_2]$
  where $\omega \in \Omega, t_1, t_2 \in \mathbb{R} \cup \{\infty\}$. This is a process which performs a data operation; t_1, t_2 are lower and upper bounds on its computation time.

- $\gamma \rightarrow P$
  where $\gamma \in \Gamma$ and P is a process term. This is to be interpreted as the process which evaluates the guard γ then if *true*, performs P.

- $P; P'$
  where P and P' are process terms: the process which performs P then (when P terminated) performs P'.

- $P + P'$
  where P and P' are process terms: the process which chooses non-deterministically to behave either as P or as P'.

- $P[> P'$
  where P and P' are process terms. This process behaves as P to begin with but P' may interrupt it at any time before it terminates, in which case P is aborted and P' runs with the network and data state inherited from P.

- $P|P'$
  where P and P' are process terms: the process consisting of P and P' running in parallel.

- X
  a process variable.

- $recX.P$
  where X is a process variable and P a process term. This captures the idea of a *recursive* process, as explained below. If X is a free variable in the term P, then the *quantifier* $recX$ binds it, and it will be seen that $recX.P$ is in a sense equivalent to the result of substituting $recX.P$ for X in P.

The semantics of these process terms is given by a body of formal rules, to be described shortly. The symbols for combining process terms bind in order of precedence as follows: $\rightarrow(high)$, ;, +, $[>$, rec, $|(low)$. Combinators of equal precendence associate to the left. Parentheses may be used to override this precedence.

Operational Semantics of bCANdle A formal model - a bCANdle system - is defined to be a triple (P, N, D) consisting of a process term, a network and a data environment. These are assumed to agree on channel identifiers, message identifiers, data variables, operation and predicate names. It is also assumed that if P has a subterm $k!i.x$, then the network semantic rules will accommodate this: whenever $v \in D.type(x)$, then $i.v \in$ the message set of N_k. Similarly, it is assumed that if P includes the subterm $k?i.x$, any message i.v receivable by N (i.e., in the message set of N_k) has $v \in D.type(x)$.

The *operational semantics* of bCANdle is defined in terms of a labelled transition system whose *states* are all bCANdle triples (P, N, D) over given $K, I, Var, \Omega, \Gamma$. Transitions between these states, $(P, N, D) \xrightarrow{\lambda} (P', N', D')$ are labelled by three kinds of labels λ:

- time passage: $t \in \mathbb{R} = \{t \in \mathbb{R} : t \geq 0\}$;
- a network action $\lambda_n \in A_n$ as defined in the previous subsection;
- a *process action label* $\lambda_p \in A_p = \Omega \cup \Gamma \cup \{k!i.v : k \in K \wedge i \in I \wedge v \in V\} \cup \{k?i.v : k \in K \wedge i \in I \wedge v \in V\}$. The process actions are thus: computations, evaluation of guards, data sends, data receives.

The timed transition system *of a bCANdle system* (P_0, N_0, D_0) is defined to be that part of the labelled transition system that is reachable from the initial state (P_0, N_0, D_0).

The transition relation is the smallest set closed under the rule

$$\frac{}{(\surd, N, D) \xrightarrow{0} (\surd, N, D)} \quad \text{(P-)}$$

and also rules set out below. $\surd$ denotes an "ideal" process term - a process which has nothing to do except terminate[1]. Variable λ_n ranges over network discrete actions A_n, t ranges over times $\mathbb{R}_{\geq 0}$; λ_{nt} over network actions $A_n \cup \mathbb{R}_{\geq 0}$; λ_p over process actions as just described; and λ over $A_p \cup A_n \cup \mathbb{R}_{\geq 0}$.

[1] cf [79], pp197-8

4.3.1 Rules for Atomic Process Terms

$$\frac{N_k = (s, u) \wedge v = D.x}{(k!i.x, N, D) \xrightarrow{k!i.v} (\surd, N[k := (s, u \leftarrow\!\!\rho\ i.v)], D)} \quad \text{(Snd 1)}$$

$$\frac{N \xrightarrow{\lambda_n}_n N'}{(k!i.x, N, D) \xrightarrow{\lambda_n} (k!i.x, N', D)} \quad \text{(Snd 2)}$$

$$\frac{}{(k!i.x, N, D) \xrightarrow{0} (k!i.x, N, D)} \quad \text{(Snd 3)}$$

$$\frac{N_k = (\uparrow i.v, u)}{(k?i.x, N, D) \xrightarrow{k?i.v} (\surd, N, D[x := v])} \quad \text{(Rcv 1)}$$

$$\frac{N \xrightarrow{\lambda_n}_n N' \wedge (\forall v \forall u. N_k \neq (\uparrow i.v, u) \vee N_k = N'_k)}{(k?i.x, N, D) \xrightarrow{\lambda_n} (k?i.x, N', D)} \quad \text{(Rcv 2)}$$

$$\frac{N \xrightarrow{t}_n N'}{(k?i.x, N, D) \xrightarrow{t} (k?i.x, N', D)} \quad \text{(Rcv 3)}$$

$$\frac{D \xrightarrow{\omega}_d D'}{([\omega : 0, t], N, D) \xrightarrow{\omega} (\surd, N, D')} \quad \text{(Comp 1)}$$

$$\frac{N \xrightarrow{\lambda_n}_n N'}{([\omega : t_1, t_2], N, D) \xrightarrow{\lambda_n} ([\omega : t_1, t_2], N', D)} \quad \text{(Comp 2)}$$

$$\frac{N \xrightarrow{t}_n N' \wedge t \leq t_2}{([\omega : t_1, t_2], N, D) \xrightarrow{t} ([\omega : t'_1, t'_2], N', D)} \quad \text{(Comp 3)}$$

where $t'_1 = max(0, t_1 - t)$ and $t'_2 = t_2 - t$

4.3.2 Rules for Guard

$$\frac{D \models \gamma}{(\gamma \rightarrow P, N, D) \xrightarrow{\gamma} (P, N, D)} \quad \text{(Gu 1)}$$

$$\frac{N \xrightarrow{\lambda_n}_n N'}{(\gamma \rightarrow P, N, D) \xrightarrow{\lambda_n} (\gamma \rightarrow P, N', D)} \quad \text{(Gu 2)}$$

$$\frac{N \xrightarrow{t}_n N' \wedge (D \not\models \gamma \vee t = 0)}{(\gamma \rightarrow P, N, D) \xrightarrow{t} (\gamma \rightarrow P, N', D)} \quad \text{(Gu 3)}$$

4.3.3 Rules for Sequential Composition

$$\frac{(P, N, D) \xrightarrow{\lambda} (P', N', D') \wedge \neg(P' \equiv \surd)}{(P; Q, N, D) \xrightarrow{\lambda} (P'; Q, N', D')} \quad \text{(Seq 1)}$$

$$\frac{(P, N, D) \xrightarrow{\lambda_p} (\surd, N', D')}{(P; Q, N, D) \xrightarrow{\lambda_p} (Q, N', D')} \quad \text{(Seq 2)}$$

4.3.4 Choice

$$\frac{(P, N, D) \xrightarrow{\lambda_p} (P', N', D')}{(P + Q, N, D) \xrightarrow{\lambda_p} (P', N', D')} \quad \text{(Ch 1)}$$

$$\frac{(Q, N, D) \xrightarrow{\lambda_p} (Q', N', D')}{(P + Q, N, D) \xrightarrow{\lambda_p} (Q', N', D')} \quad \text{(Ch 2)}$$

$$\frac{(P, N, D) \xrightarrow{\lambda_{nt}} (P', N', D) \wedge (Q, N, D) \xrightarrow{\lambda_{nt}} (Q', N', D)}{(P + Q, N, D) \xrightarrow{\lambda_{nt}} (P' + Q', N', D)} \quad \text{(Ch 3)}$$

4.3.5 Recursion

Where $P[Q/X]$ means the process term resulting from uniform substitution of the term Q for the process variable X,

$$\frac{(P[recX.P/X], N, D) \xrightarrow{\lambda} (P', N', D')}{(recX.P, N, D) \xrightarrow{\lambda} (P', N', D')} \quad \text{(Rec)}$$

4.3.6 Interrupt

$$\frac{(P, N, D) \xrightarrow{\lambda_p} (P', N', D') \wedge \neg(P' \equiv \surd)}{(P[> Q, N, D) \xrightarrow{\lambda_p} (P'[> Q, N', D')} \quad \text{(Int 1)}$$

$$\frac{(P,N,D) \xrightarrow{\lambda_p} (\surd, N', D')}{(P[> Q, N, D) \xrightarrow{\lambda_p} (\surd, N', D')} \quad \text{(Int 2)}$$

$$\frac{(Q,N,D) \xrightarrow{\lambda_p} (Q', N', D')}{(P[> Q, N, D) \xrightarrow{\lambda_p} (Q', N', D')} \quad \text{(Int 3)}$$

$$\frac{(P,N,D) \xrightarrow{\lambda_{nt}} (P', N', D) \wedge (Q,N,D) \xrightarrow{\lambda_{nt}} (Q', N', D)}{(P[> Q, N, D) \xrightarrow{\lambda_{nt}} (P'[> Q', N', D)} \quad \text{(Int 4)}$$

4.3.7 Parallel Composition

$$\frac{(P,N,D) \xrightarrow{\lambda_p} (P', N', D') \wedge \neg(P' \equiv \surd)}{(P|Q, N, D) \xrightarrow{\lambda_p} (P'|Q, N', D')} \quad \text{(Par 1)}$$

$$\frac{(P,N,D) \xrightarrow{\lambda_p} (\surd, N', D')}{(P|Q, N, D) \xrightarrow{\lambda_p} (Q, N', D')} \quad \text{(Par 2)}$$

$$\frac{(Q,N,D) \xrightarrow{\lambda_p} (Q', N', D') \wedge \neg(Q' \equiv \surd)}{(P|Q, N, D) \xrightarrow{\lambda_p} (P|Q', N', D')} \quad \text{(Par 3)}$$

$$\frac{(Q,N,D) \xrightarrow{\lambda_p} (\surd, N', D')}{(P|Q, N, D) \xrightarrow{\lambda_p} (P, N', D')} \quad \text{(Par 4)}$$

$$\frac{(P,N,D) \xrightarrow{\lambda_{nt}} (P', N', D) \wedge (Q,N,D) \xrightarrow{\lambda_{nt}} (Q', N', D)}{(P|Q, N, D) \xrightarrow{\lambda_{nt}} (P'|Q', N', D)} \quad \text{(Par 5)}$$

4.3.8 A Note on Recursion

The process term $recX.P$ captures the idea of a *recursive* process. If X is a free variable in the term P, then the *quantifier* $recX$ binds it, and $recX.P$ represents is a process that is "equivalent" to the result of substituting $recX.P$ uniformly for X in P. This is effectively the meaning of rule (Rec) and the "equivalence" can be formally derived from this rule as a strong

bisimulation.

To see how this captures the the idea of recursion, think of a process which iterates an action a (a computation $[\omega : t_1, t_2]$ for instance). This could be defined as "A where $A \triangleq a; A$": a recursive definition, since A appears in the right hand side. In terms of the present formalism, this recursively defined iterating process would be written $recA.(a; A)$. The equivalence then asserts that this process is equivalent to a;$recA.(a; A)$ which has just the semantics needed for iterating a.

Similarly, a process A which iterates actions a, b alternately, in an endless loop would be defined recursively by means of a pair of equations:

$$A \triangleq a; B$$
$$B \triangleq b; A$$

A is defined to be a process which consists in doing a then behaving like B, while B consists in doing b then behaving like A. A is succinctly defined to be $recA.(a; recB.(b; A))$.

The *rec* construct essentially presents recursion as a "fixed point" in the sense that R Milner explains in [98], p 57.

4.3.9 Equational Laws

Following Milner [98], bCANDle process terms can be given an "equational" presentation ([79]):

$$P + Q = Q + P \quad (+1)$$

$$P + (Q + R) = (P + Q) + R \quad (+2)$$

$$P + P = P \quad (+4)$$

$$P + idle = P \quad (+5)$$

$$P; (Q; R) = (P; Q); R \quad (;1)$$

$$(P + Q); R = P; R + Q; R \quad (;2)$$

$$idle; P = idle \quad (;3)$$

$$idle[> P = P \quad (I1)$$

$$P[> idle = P \quad (I2)$$

$$P[> (Q[> R) = (P[> Q)[> R \quad (I3)$$

$$P|Q = Q|P \quad (P1)$$

$$P|(Q|R) = (P|Q)|R \quad (P2)$$

$$P|idle = P \text{ if } P \text{ is persistent} \quad (P3)$$

$$recX.P = P[recX.P/X] \quad (R1)$$

$$P[Q/X] = Q,\ X \text{ guarded in } P \Rightarrow recX.P = Q \quad (R2)$$

Here, $idle \triangleq [ID : \infty, \infty]$ where $ID \in \Omega$ is a symbol for an operation that does nothing; thus *idle* is a process that does nothing, indefinitely!

X guarded in P means that process variable X occurs in P in subterm(s) of the form $P_1; P_2$ where the occurrence of X is in P_2 and P_1 *is guarding* – that is, P_1 is atomic or is a sequence of parallel composite of terms one of which is guarding, or is a "+" or "[>" combination of terms which are both guarding.

The point of the set of equational laws is that one can define a relation $\vdash P = Q$ meaning "equality of P, Q is derivable from the equational laws". This gives a notion of "equality" of process terms beyond syntactic identity,

which is *sound*: $\vdash P = Q$ implies that P, Q are *semantically equivalent* in a sense which will be made precise in the discussion of *strong bisimulation* and *strong equivalence* below.

Chapter 5

Modelling bCANdle with Automata

5.1 Introduction

Having defined a timed transition system semantics for bCANdle as outlined above, Kendall developed a formalism for representing a bCANdle system as a timed automaton and showed this automaton's semantics to be strongly equivalent to it. This construction is monolithic and computationally expensive. The automaton constructed from a bCANdle system does not naturally resolve into components and is apt to have a very large state spece.

The present chapter develops an adaptation of this formalism in which the CAN network channels are modelled as timed automata and these composed in parallel with automata representing process terms and associated data. This will facilitate a *compositional* view of the CAN-based system, so mitigating the state space explosion problem.

First, an overview of the model checking tool UPPAAL, and the extended timed automata modelled by UPPAAL is presented; then an UPPAAL model of a CAN channel is presented, followed by a simple examples of its use.

After this a mapping of bCANdle systems to parallel products of timed automata is developed, and shown to induce a strong bisimulation between the timed transition systems of the two formalisms. Thus, the models de-

veloped in this chapter are semantically equivalent to the bCANdle models from which they are derived: they exhibit equivalent behaviour.

5.2 UPPAAL and Timed Automata

Once we have developed a model of bCANdle as a parallel product of timed automata, we wish to be able to check its properties. The UPPAAL tool has turned out to be useful for this as it supports all the timed-automata constructs, in the extended sense of §2.2.4. A recent tutorial overview of UPPAAL is given by G Behrmann, A David and K G Larsen [19] (updating earlier accounts [50][89]) and a discussion of its theoretical underpinnings by J Bengtsson and W Yi [23]. Parallel products of timed automata are programmable as UPPAAL *systems* in a natural way. In this section we review UPPAAL syntax and semantics. This mirrors the time-automata semantics faithfully: we need to see that a UPPAAL system *simulates* a parallel product of timed automata in the sense of §2.4.2.

5.2.1 UPPAAL Syntax

This is given in full in the "help" that comes with the UPPAAL tool. The following summary deals with the small subset of UPPAAL features employed in the present work.

An UPPAAL *system* comprises three sections.

- First, a section of global declarations: variables and constants scoped over the whole system are declared using a C++-like syntax (for instance, `const int a = 5;`). User-defined types and functions may also be declared although these will not play a role in the present investigation.

  Arrays and `structs` may also be declared and optionally initialised, again with C++ style syntax: for example, `int b[3] = {3, 5, 7};` for an array of three integers.

Subranges of integer type may be declared: for example `int[0,9] n;` is an integer variable with values in the range 0 to 9.

`clock x;` simply declares a clock variable. It is initialised to 0 at the beginning of a run and whenever a transition calls for it to be reset.

`chan ch;` declares a *binary channel*. Edges in two automata can be then given respective *synchronisation labels* `ch!`, `ch?`, forcing them to synchronise, producing a joint or "rendez-vous" transition just as in the definition of the parallel product of timed automata. A channel may be declared `urgent`, slightly altering the semantics (see below). A `broadcast channel` may be used to provide one-to-many synchronisation: one edge is labelled `ch!` and two or more may be labelled `ch?`.

- The second section comprises one or more *process templates*, each headed `process`, followed by a process name and a list of formal parameters which are declarations as above. A process template defines a timed automaton *type*; the automata comprising the system are *instances* of process templates.

  Within a template are declarations of *local variables*, followed by declarations of all the *locations* and *edges* of the automaton.

  First, after the key word `state` is a list of all the locations. Each location in the list may have an *invariant* listed after it in braces {}.

  Next, after the key word `commit` is a list of the locations which are *committed*, then, after the key word `urgent`, a list of the *urgent* locations, then after the key word `init`, the *initial location* of the automaton. Committed and urgent locations are explained in the semantics section below.

  After this, and after the key word `trans` is a comma-separated list of all the edges of the automaton. Each edge is declared as *source location* `->` *target location* followed optionally by, in braces {},
  – key word `guard` followed by a *guard* for the edge, and a semicolon,
  – key word `sync` followed by a *synchronisation label* which must be of

the form c! or c? where c is a channel, a semicolon, and
– key word `assign` followed by a list of *assignment* or update statements.

The update statements have a fairly self-evident C++-style syntax. They may be compound statements with function calls, conditionals and iterators (including a Java-style `for(x:lst)`) but for the purposes of the present work, simple assignment statements suffice: `x++, y -= 2, z = x * 2` and so forth. For backwards compatibility, a Pascal-style assignment operator `:=` is also permitted. Several update statements may appear in a comma-separated list terminated by a semicolon.

- The third section defines the *processes* – *instances* of the process templates defined in the second section. Several instances of one template may be defined. The syntax of a process definition is an instance name, an assignment operator, a template, a list in parentheses () of parameters conforming to the formal parameters of the template definition, and a semicolon. Finally, there is a *system definition* – a simple list of all the processes comprising the system.

Version 4 of UPPAAL also provides syntax to declare *priorities* between synchronisation channels and processes, to resolve non–determinism when more than one action or synchronisation is enabled at once, and *meta-variables* which can be used to track verifications without being saved in the system state. The present work does not employ either of these features.

5.2.2 UPPAAL Semantics

The semantics of UPPAAL are based on timed transition systems just as previously defined, but there are extra conditions defining *urgency*, *committedness* and the precise nature of the synchronisations. The semantic rules described in this section are paraphrased from the UPPAAL 4.0 help system; see also Behrmann *et al* [19].

An *urgent location* is easiest to understand in terms of standard timed automata: imagine an extra clock x, reset on every edge running into the location, and the location has invariant $x \leq 0$. Thus, no time may elapse at an urgent location.

A *committed* location is not only urgent in this sense, but in any run of the timed transition system, wherever the state (not the last in the run) includes a committed location, the *next* action transition *must* be from a committed location. Thus, if there is just one committed location in a state, the actions leading to and from the committed location are, in effect, *atomic*.

The timed transition system of an UPPAAL system is of the usual kind: see for instance §§2.2.2-2.2.4. A *state* $(\overrightarrow{l}, v)$ is a vector $\overrightarrow{l} = (l_1, ..., l_n)$ of component process (automaton) locations together with a valuation v of all clocks and variables. Transitions between states are *delay transitions* or *action transitions*.

A delay transition is of the form $(\overrightarrow{l}, v) \xrightarrow{d} (\overrightarrow{l}, v)$ where d is a non-negative real number, a time delay, and $v + d$ is the valuation for which $(v + d)(x) = v(x) + d$ for any clock x, and which agrees with v on all other variables. This delay transition is permitted only if

- for every $d' \in [0, d]$, $v + d'$ satisfies the invariant of $\overrightarrow{l}$, the conjunction of all the component invariants,
- no l_i is committed or urgent,
- no edge from any l_i synchronises on an urgent channel, or if it does, its guard is *not* satisfied by $v + d'$ for any $d' \in [0, d]$.

An action transition is one of three kinds – *internal* transitions, *binary synchronisations* and *broadcast* synchronisations.

- An internal transition has the form $(\overrightarrow{l}, v) \rightarrow (\overrightarrow{l}', v')$ where in some component process k, there is an edge $l_k \xrightarrow{\zeta,\lambda} l'_k$ with no synchronisation label, v satisfies the guard ζ, $l'_i = l_i$ for $i \neq k$, and valuation v' is as v *after* the updates (resets, assignments) in set λ have been executed.

  It is further required that v' satisfy the invariants of $\overrightarrow{l}'$, that l_k is committed or no other l_i is committed, and that no other action transition

from $(\overrightarrow{l}, v)$ has higher priority. The last proviso can be ignored as we do not use priorities in this work.

- A binary synchronisation has the form $(\overrightarrow{l}, v) \rightarrow (\overrightarrow{l}', v')$ where in some component process j, there is an edge $l_j \overset{\zeta_j, c!, \lambda_j}{\longrightarrow} l'_j$ with synchronisation label `c!` for some channel `c`, and in another component process k, there is an edge $l_k \overset{\zeta_k, c?, \lambda_k}{\longrightarrow} l'_k$ with synchronisation label `c?`. The rest of $\overrightarrow{l}'$ is like $\overrightarrow{l}$: $l'_i = l_i$ for $i \neq j, k$.

  It is required that $v \models \zeta_j \wedge \zeta_k$.

  v' is the valuation obtained from v by *first* executing updates λ_j *then* executing updates λ_k. It is required that this v' satisfy the invariants of $\overrightarrow{l}'$.

  It is further required that if neither of $l_j,\ l_k$ is committed, neither is any other component location l_i, and that no other action transition from $(\overrightarrow{l}, v)$ has higher priority (again, this can be ignored for the present purpose).

- A broadcast synchronisation has the form $(\overrightarrow{l}, v) \rightarrow (\overrightarrow{l}', v')$ where in some component process j, there is an edge $l_j \overset{\zeta_j, c!, \lambda_j}{\longrightarrow} l'_j$ with synchronisation label `c!` for some *broadcast* channel `c`, and in m component processes $k_1, ..., k_m$ edges $l_{k_p} \overset{\zeta_{k_p}, c?, \lambda_{k_p}}{\longrightarrow} l'_{k_p}$ $(p = 1, ..., m)$, with synchronisation label `c?`. The m synchronisation-receiving edges belong to *different* processes. In what follows, it is assumed the numbering of processes $k_1, ..., k_m$ is in the order the processes are mentioned in the system definition.

  It is required that v satisfies all the guards: $v \models \zeta_j \wedge \zeta_{k_1} \wedge ... \wedge \zeta_{k_m}$.

  For all locations l_i in $\overrightarrow{l}$ *not* one of $l_j, l_{k_1}, ..., l_{k_m}$, *no* edge from l_i has a synchronisation label `c?` or if it does, v does *not* satisfy its guard. For all of these locations, $l'_i = l_i$, to complete the definition of $\overrightarrow{l}'$.

  v' is the valuation obtained from v by *first* executing updates λ_j *then* executing updates λ_{k_p} in order of $p = 1, 2, ..., m$.

  It is required that this v' satisfy all the invariants of $\overrightarrow{l}'$.

If none of $l_j,\ l_{k_1},\ \ldots\ l_{k_m}$ is committed, neither is any other component location l_i of $\overrightarrow{l}$, and no other action transition from $(\overrightarrow{l}, v)$ has higher priority (again, priorities can be ignored for the present purpose).

5.3 A CAN channel

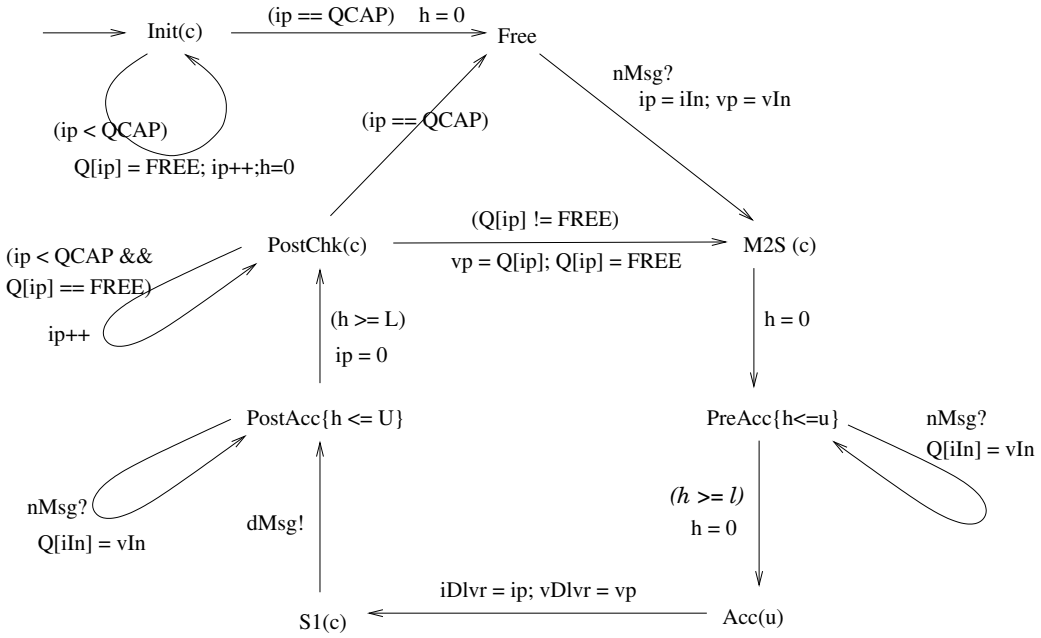


Figure 5.1: Automaton to model a CAN Channel

Figure 5.1 shows the locations and transitions of an UPPAAL timed automaton representing a CAN channel. Note that the legend (c) denotes a *committed* location in the UPPAAL sense (see above). The UPPAAL source code for this appears below in §5.3.2, in the definition of the `process Channel` template. `QCAP` is a parameter denoting the number of distinct message types supported by the channel. In order to function, the automaton is equipped with some *local* and some *global* variables, the latter shareable with other automata that might be composed with the CAN channel. The local variables include a clock `h` to track its behaviour in time, and integer variables `ip` and `vp` to represent the message identifier ("priority") and

message pay-load respectively of the message "currently" in process of being transmitted through the CAN channel.

Global variables are needed to interface this model with sending process(es) and receiving process(es). Specifically, there are integers `iIn` and `vIn` shared with a sending process to represent the identifier and data pay-load of a message to be sent; integers `iDlvr` and `vDlvr` shared with a receiving process to represent the identifier and data pay-load of a message to be handed on to the receiving process. The *local* array `Q[]` of integers, one entry for each distinct message identifier value, comprises the "buffers" of the CAN channel. The data type is `int` in this model but in reality would be the type of the data pay-load.

Last but not least are two UPPAAL "channels", `urgent chan nMsg` and `broadcast chan dMsg`, not to be confused with the CAN channel we are modelling. These are declared as formal parameters to the CAN channel model automaton because they are shared between this and the sending and receiving processes. Their function is to provide appropriate synchronisation of the sending process with the channel model and of the latter with 0 or more receiving processes. In particular a sending process sends a message by setting shared variables `iIn, vIn` and then offering synchronization on `urgent chan nMsg`. This is a synchronisation in the usual timed-automaton sense, with the particular feature that the synchronised action occurs *as soon as* guards on the component processes determine that it is able. The channel automaton hands a message to any receiving process by setting shared variables `iDlvr, vDlvr` to `ip, vp` and then offering synchronization `broadcast chan dMsg`. Again, this is a synchronisation in the "usual" sense, but an UPPAAL broadcast channel will synchronise with as many parallel processes as are in a position to do so. Thus, the channel hands the CAN message to as many receiving processes as care to take delivery of it; the message is not available to processes at any later time.

The automaton starts in location `Init` where it initialises each entry of its "buffer" array to the special value `FREE`, meaning there is no message in the buffer. Constant `QCAP` is the number of distinct message identifier values or priorities: the number of entries in the buffer array.

Once this has occurred, the automaton is forced to location `Free` where it waits an arbitrarily long time for a `nMsg` synchronization, upon which the assignments `ip := iIn, vp := vIn` store the incoming message. It is necessary for `nMsg` to be an `urgent chan` because we do not want the CAN channel automaton to have the option of sitting at location `Free` while a sending process is offering synchronization – waiting to send.

The process of sending the message starts immediately after this. At location `Free`, the queue of messages is empty; so a message acquired by the channel from location `Free` is presumably the correct one to send. There is an immediate transition (with a clock reset) to `PreAcc`, marking the beginning of the *pre-acceptance phase* of the message, as defined by [79].

During this phase, at the location `PreAcc`, the self-loop allows more messages to enter the channel; these are stored in the priority queue `Q[]`. It is permitted for a message with id (priority) `i` to enter a CAN channel overwriting a message `i` already there but we do not expect this to destroy a message once it is in process of being transmitted; and it does not as the latter is saved in variables `ip, vp`.

The pre-acceptance phase of CAN transmission lasts a period of time bounded by `l` and `u`, parameters of the CAN channel. The local clock `h` times this.

After the pre-acceptance phase is the *acceptance point*, represented here by an *urgent location* `Acc`.

This is followed by the *post-acceptance phase*, also timed by `h` and lasting a time bounded by parameters `L` and `U`. Post-acceptance is where the message is handed on to the receiving process(es). Note the use of a committed location to ensure correct sequencing: *first* the shared variables `iDlvr` and `vDlvr` are set, *then* synchronization on the `broadcast chan` is offered.

While in the post-acceptance phase, one or more additional messages be enter the channel. This is modelled by the self-loop at location `PostAcc`.

The transition to `PostChk` ensures that if there is another message in any of the buffers, `Q[]`, it is sent right away. The channel only returns to the "idle" state `Free` if *all* the buffers are Free.

The self-loop at committed location `PostChk` searches all the buffers for a

message (`Q[ip] != FREE`). If such exists, the highest priority (lowest index) one is sent: this is the transition to location `M2S`. Otherwise, if all the buffers are FREE (which might happen on transition from `PostAcc`, after a full circuit of the duty cycle), the automaton transits back to `Free` to wait for a message to send.

It should be noted that a couple of assumptions have been made about the behaviour of a CAN channel. First, it is possible for an arrival of a message at priority `p` to be followed by the arrival of another message at the same priority, before the first message has been accepted. In this case, the second message overwrites the first. This appears to be permitted behaviour of a CAN channel.

Second, and assumption of *homogeneity* is made: that the parameters `u, l, U, L` have the same values at all nodes capable of sending on a particular channel.

5.3.1 A Simple Example

This model of a CAN channel is simply illustrated by employing it in a simple example of a flow regulator (cf. [79], p 89) described in bCANdle as a parallel combination of process terms `Flow | Valve` where

- `Flow` is defined by the equation

```
Flow = [ReadSensor:85,90] ; k!flow.x;
       idle [> [PERIOD:10000,10250] ;
       Flow
```

– a process which periodically reads a sensor and transmits the reading onto a network channel as a message of type `flow`, and

```
Valve = k?flow.y ; [AdjustValve:200,300] ;
        Valve
```

– a process which repeatedly waits for a message of this type and on receiving one, uses it to adjust a valve.

The network part of this system consists of a single channel with bCANdle description `(flow:1, 45, 55, 10, 12)` specifying a single message type and values of l, u, L, U respectively. The data model part a variable x ("belonging" to the `Flow` process) and a variable y (the `Valve` process).

The two process specifications are *equational presentations* of the process terms

```
rec X.([ReadSensor:85,90] ; k!flow.x;
  (idle [> [PERIOD:10000,10250]) ; X)
```

and

```
rec Y.(k?flow.y ; [AdjustValve:200,300] ; Y)
```

An informal interpretation of this system is as a parallel composition of one instance of the channel automaton, figure 5.1 with the automata shown in figures 5.2 and 5.3.

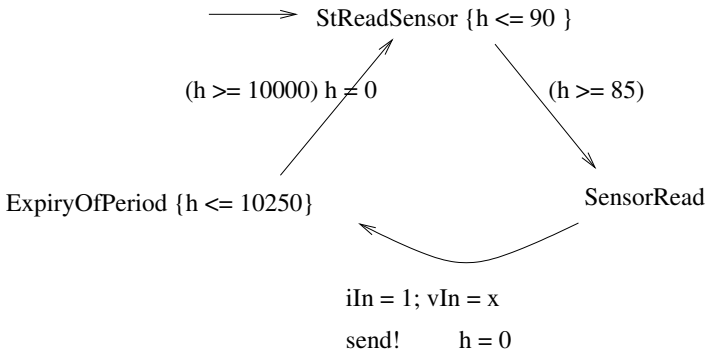


Figure 5.2: Automaton to model the Flow process

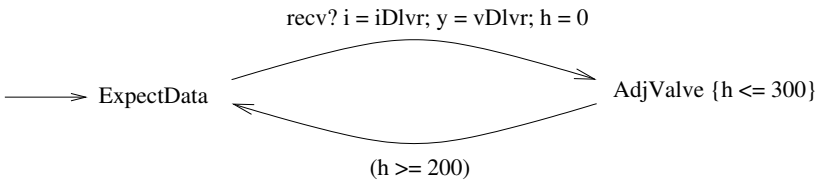


Figure 5.3: Automaton to model the Valve process

The derivation of these automata from the corresponding bCANdle process terms will be examined in more detail in the next section.

5.3.2 Example – UPPAAL Code

The following is the source code for an UPPAAL system modelling the example of §5.3.1, including the CAN channel model. After some global declarations, three process templates are defined, an instance of each is defined, and a system defined comprising these three instances.

```
clock globalClock;
const QCAP 10, FREE -1;
int vIn := 0, iIn := 0; /* input message payload */
int vDlvr := 0, iDlvr;
broadcast chan dlvrMsg;
urgent chan newMsg;

process Channel(const l; const u; const L; const U;
                urgent chan nMsg; broadcast chan dMsg) {
  clock h;
  int ip, vp;
  int Q[QCAP+1];

  state Free, PreAcc{h <= u}, Acc, PostAcc{h <= U},
        PostChk, M2S, Init, S1;
  commit PostChk, M2S, S1;
  urgent Acc;
  init Init;

  trans
    Init -> Init {
      guard ip <= QCAP;
      assign Q[ip] := FREE, ip++;
    },
    Init -> Free {
      guard ip == QCAP+1;
      assign h := 0;
    },
```

```
    Free -> M2S {
      sync nMsg?;
      assign ip := iIn, vp:= vIn;
    },
    M2S -> PreAcc {
      assign h := 0;
    },
    PreAcc -> PreAcc {
      sync nMsg?;
      assign Q[iIn] := vIn;
    },
    PreAcc -> Acc {
      guard h >= l;
      assign h := 0;
    },
    Acc -> S1 {
      assign iDlvr := ip, vDlvr := vp;
    },
    S1 -> PostAcc {
      sync dMsg!;
    },
    PostAcc -> PostAcc {
      sync nMsg?;
      assign Q[iIn] := vIn;
    },
    PostAcc -> PostChk {
      guard h >= L;
      assign ip := 0;
    },
    PostChk -> Free {
      guard ip == QCAP;
    },
    PostChk -> PostChk {
      guard ip < QCAP && Q[ip] == FREE;
      assign ip++;
    },
    PostChk -> M2S {
      guard ip < QCAP && Q[ip] != FREE;
      assign vp := Q[ip], Q[ip] := FREE;
    };
}
```

```
process Flow(urgent chan c) {
  clock h, k;

  state ReadSensor{h <= 90}, StReadSensor{h <= 10250}, SensorRead;
  init StReadSensor;
  trans
    StReadSensor -> ReadSensor {
      guard h >= 10000;
      assign h := 0, k := 0, iIn := 1;
    },
    ReadSensor -> ReadSensor {
      guard k >= 5;
      assign vIn++, k := 0;
    },
    ReadSensor -> SensorRead {
      guard h >= 85;
    },
    SensorRead -> StReadSensor {
      sync c!;
      assign h := 0;
    };
  }

process Valve(broadcast chan c) {
  clock m;
  int i, y;

  state expectMsg, AdjustValve{m <=300};
  init expectMsg;

  trans
    expectMsg -> AdjustValve {
      sync c?; assign m := 0, i := iDlvr, y := vDlvr;
    },
    AdjustValve -> expectMsg {
      guard m >= 200;
    };
  }

theChannel := Channel(45, 55, 10, 15, newMsg, dlvrMsg);
```

```
flow := Flow(newMsg);
valve := Valve(dlvrMsg);
system flow, theChannel, valve;
```

5.3.3 A Broadcast Example

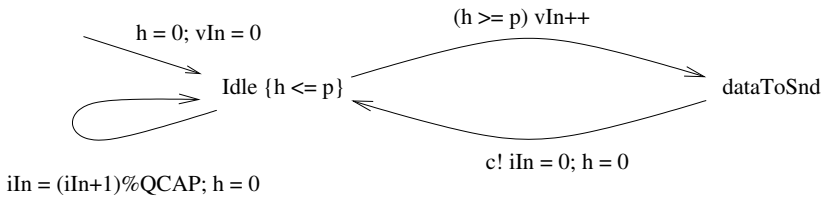


Figure 5.4: An abstract producer process

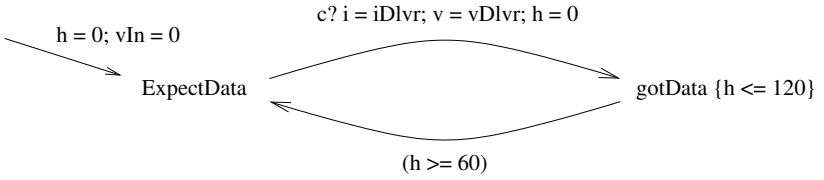


Figure 5.5: An abstract consumer process

The CAN channel model presented here exhibits *broadcast* semantics if *broadcast channels*, as devised in the UPPAAL modelling tool (see §5.2.2), are incorporated in the compositional scheme. To illustrate this, consider an UPPAAL system comprising the CAN channel model accepting messages from the abstract producer process illustrated by figure 5.4 and several instances of the abstract consumer process illustrated in figure 5.5. Again, channel `c` of the producer is bound to a globally defined urgent channel `newMsg` and the channel `c` of the consumer(s) to a globally defined broadcast channel `dlvrMsg`, to which the channel parameters of the CAN model are also bound.

If one experiments with varying numbers of instances of the consumer, it is found that indeed the CAN channel model's behaviour is to *broadcast*: `dlvrMsg!` is capable of synchronizing with 0, 1 or several `dlvrMsg?` edges.

5.4 A Compositional Model of bCANdle

The task now is to define a formal translation of bCANdle into a system of parallel timed automata whose semantics are strongly equivalent to those of the original bCANdle system. The approach is a modification of Kendall's *clocked bCANdle system* construction ([79], chapter 4).

The basic idea is to build, for each process in a bCANdle system, a timed automaton whose timed transition system will mirror behaviour of the process within the bCANdle system. The locations of the automaton are pairs consisting of a process term and an associated data environment. The process term is to be thought of intuitively as a "to do" list: a transition of the automaton from, say (P, D) to (P', D') is possible when P' is "what is to do" after an action has occurred starting from P and the data model has in consequence evolved from D to D'. This is a common way to interpret process algebra and is implicit in the operational semantics of section 4.3.

However, here we are defining not a timed transition system but an automaton whose timed semantics will mirror the bCANdle timed semantics. To bring this about, in the process terms, atomic operations $[\omega : t_1, t_2]$ are augmented with a clock h each: $[\omega : t_1, t_2]^h$ and a transition $([\omega : t_1, t_2]^h; Q, D) \longrightarrow (Q, D')$ is guarded by the condition $h \geq t_1$ while location $([\omega : t_1, t_2]^h; Q, D)$ must satisfy the invariant $h \leq t_2$.

The formal definitions follow. First, fix a set of data values and variables, a set Γ of predicate symbols over the variables, and a set Ω of operation symbols over the variables, over which a data environment may be defined as in section 4.1. Suppose given a bCANdle system (P, N, D) in which the process term $P = P_1|...|P_n$ where the P_i are bCANdle process terms *not* containing the parallel composition symbol. It is also assumed that time constraints on atomic operations (e.g. $[\omega : t_1, t_2],\ \omega \in \Omega$) are natural number-valued, compound terms have *static control* and clocks are allocated *safely* in the sense explained in §5.4.1 below.

By static control, we mean that no process term has a subterm of the form $recX.P'$ in which P' has a subterm of the form $Q; R$ or $Q[> R$ and the process variable X occurs free in Q. This, as [79] explains, ensures that $recX.P'$ does

give rise to unbounded recursion, and that the timed automaton shortly to be described has a finite number of states.

We further assume that the sets of variables occurring in the P_i are disjoint and that hence $D = D_1 \cup ... \cup D_n$ where D_i is the restriction of D to the variables in P_i. Here the restriction of a data model is the data model made by restricting the four functions making it up, and the union of two data models with disjoint variables sets is the pairwise set-theoretic union of the four constituent functions as sets.

5.4.1 Clocked Process Terms

This is defined as in ([79], §4.2), but is restricted to process terms *not* employing parallel composition |. A clocked process term is defined recursively as follows:

- A bCANdle send or receive process term, $k!i.x$ or $k?i.x$ is a clocked process term;
- Where $[\omega : t_1, t_2]$ is a basic process term representing an operation, and t_1, t_2 are natural numbers or ∞, and h a new clock variable, $[\omega : t_1, t_2]^h$ is a clocked process term.
- Where γ is a predicate symbol, X a process variable and $\hat{Q}, \hat{Q}'$ clocked process terms, so are X, $\gamma \rightarrow \hat{Q}$, $recX.\hat{Q}$, $\hat{Q};\hat{Q}'$, $\hat{Q} + \hat{Q}'$, $\hat{Q}[> \hat{Q}'$.

The set $H_{\hat{Q}}$ of *initial clock variables* of the clocked process term $\hat{Q}$ is defined recursively by:

- If $\hat{Q}$ is a send or receive term, then $H_{\hat{Q}} = \emptyset$;
- If $\hat{Q}$ is $[\omega : t_1, t_2]^h$, then $H_{\hat{Q}} = \{h\}$;
- $H_{\gamma \rightarrow \hat{Q}} = \emptyset$;
- $H_{\hat{Q};\hat{Q}'} = H_{\hat{Q}}$;
- $H_{\hat{Q}+\hat{Q}'} = H_{\hat{Q}[>\hat{Q}'} = H_{\hat{Q}} \cup H_{\hat{Q}'}$.

A clocked process term is *safely* clocked, the allocation of clocks to the term is *safe*, if in every subterm of the form $\hat{Q}[> \hat{R}$, $\hat{R}$ is guarded (every process variable is guarded in the sense of §4.3.9) and the initial clock variables of $\hat{R}$ do not appear among the clock variables of $\hat{Q}$. This will ensure that in the timed automaton construction described below, reset of the clocks allocated to the different subterms of a process term do not interfere with one another. Kendall discusses a similar situation in [79], chapter 4, further requiring that in a subterm of the form $\hat{Q}|\hat{R}$, the clock variables of $\hat{Q}$ and $\hat{R}$ are disjoint; we do not require this as we do not include parallel composition in our syntax.

5.4.2 An Automaton Construction

Let $\hat{P}_0$ be a clocked process term and D_0 a data model supporting all the data variables and operations of $\hat{P}_0$. The timed automaton $\mathbb{A}(\hat{P}_0, D_0)$ is defined as follows:

- The *locations* are pairs consisting of a clocked process term and a data model supporting all the data variables and operations of the process term. Corresponding to the ideal process term $\surd$ introduced in § 4.3 there are locations $(\surd, D)$ where D is a data model. The *initial* location is $(\hat{P}_0, D_0)$. Once the transition relation is defined we restrict to the locations reachable from $(\hat{P}_0, D_0)$. Let Π denote the set of these locations.

- The action alphabet is the set A_p defined in section 4.3. Recall that this consists of *predicate* symbols $\gamma \in \Gamma$ for which the corresponding action is evaluation of a guard γ; *operation* symbols $\omega \in \Omega$; and CAN send and receive actions $k!i.v, k?i.v$ where i is a message identifier, v a variable, k a CAN channel.

- The transition relation is the least $E \subseteq \Pi \times Z_{\mathcal{H}} \times A_p \times 2^{\mathcal{H}} \times \Pi$ closed under the rules set out in § 5.4.2.2 below. $\mathcal{H}$ is the set of clock variables

occuring in Π and Z_H the set of *clock zones*, defined in the usual way [1], and $((\hat{Q}, D), \zeta, \lambda, H, (\hat{Q}', D')) \in E$ is as usual written

$$(\hat{Q}, D) \xrightarrow{\zeta,\lambda,H} (\hat{Q}', D')$$

- $\mathcal{H}$ includes a special clock variable h^u reset on every transition edge and which forces an *urgent* action by the conjunction of $h^u \leq 0$ with the invariant at the target location. The *invariant* function $I : \Pi \to Z_{\mathcal{H}}$ is defined in § 5.4.2.1 below.

5.4.2.1 The Invariant Function

On the ideal locations, $I(\surd, D) = (h^u \leq 0)$ (cf. [79] pp. 197-8).

I is defined on atomic clocked process terms thus. (h^u is the urgent clock, 1 the "top" element of the boolean algebra 2.)

$$\begin{aligned} I(k!i.x, D) &= (h^u \leq 0) \\ I(k?i.x, D) &= 1 \\ I([\omega : t_1, t_2]^h, D) &= (h \leq t_2) \end{aligned}$$

...and recursively on compound terms thus.

$$\begin{aligned} I(\gamma \to \hat{Q}, D) &= \begin{cases} (h^u \leq 0) & \text{if } D \models \gamma \\ 1 & \text{otherwise} \end{cases} \\ I(\hat{Q}; \hat{Q}', D) &= I(\hat{Q}, D) \\ I(\hat{Q} + \hat{Q}', D) &= I(\hat{Q}, D) \wedge I(\hat{Q}', D) \\ I(\hat{Q}[> \hat{Q}', D) &= I(\hat{Q}, D) \wedge I(\hat{Q}', D) \\ I(recX.\hat{Q}, D) &= I(\hat{Q}[recX.\hat{Q}/X], D) \end{aligned}$$

5.4.2.2 The Transition Relation

E is the smallest relation closed under the following rules.

[1] as logical conjunctions of inequalities between a clock variable or difference of two clock variables and a constant

$$\frac{}{(k!i.x, D) \overset{1,k!i.v,\{h^u\}}{\longrightarrow} (\surd, D)} \quad \text{(C\_Snd)}$$

$$\frac{}{(k?i.x, D) \overset{1,k?i.v,\{h^u\}}{\longrightarrow} (\surd, D[x := v])} \quad \text{(C\_Rcv)}$$

$$\frac{D \overset{\omega}{\longrightarrow}_d D'}{([\omega : t_1, t_2]^h, D) \overset{(h \geq t_1),\omega,\{h^u\}}{\longrightarrow} (\surd, D')} \quad \text{(C\_Cmp)}$$

$$\frac{D \models \gamma}{(\gamma \rightarrow Q, D) \overset{1,\gamma,\{h^u\} \cup H_Q}{\longrightarrow} (Q, D)} \quad \text{(C\_Gd)}$$

where $H_{\hat{Q}}$ denotes the set of *initial clock variables* as defined in §5.4.1.

$$\frac{(\hat{Q}, D) \overset{\zeta,\lambda,H}{\longrightarrow} (\hat{Q}', D') \wedge \neg \hat{Q}' \equiv \surd}{(\hat{Q}; \hat{R}, D) \overset{\zeta,\lambda,H}{\longrightarrow} (\hat{Q}'; \hat{R}, D')} \quad \text{(C\_Seq 1)}$$

$$\frac{(\hat{Q}, D) \overset{\zeta,\lambda,H}{\longrightarrow} (\surd, D')}{(\hat{Q}; \hat{R}, D) \overset{\zeta,\lambda,H \cup H_{\hat{R}}}{\longrightarrow} (\hat{R}, D')} \quad \text{(C\_Seq 2)}$$

where $H_{\hat{R}}$ denotes the set of initial clock variables as in §5.4.1.

$$\frac{(\hat{Q}, D) \overset{\zeta,\lambda,H}{\longrightarrow} (\hat{Q}', D')}{(\hat{Q} + \hat{R}, D) \overset{\zeta,\lambda,H}{\longrightarrow} (\hat{Q}', D')} \quad \text{(C\_Ch 1)}$$

$$\frac{(\hat{R}, D) \overset{\zeta,\lambda,H}{\longrightarrow} (\hat{R}', D')}{(\hat{Q} + \hat{R}, D) \overset{\zeta,\lambda,H}{\longrightarrow} (\hat{R}', D')} \quad \text{(C\_Ch 2)}$$

$$\frac{(\hat{Q}[recX.\hat{Q}/X], D) \overset{\zeta,\lambda,H}{\longrightarrow} (\hat{Q}', D')}{(recX.\hat{Q}, D) \overset{\zeta,\lambda,H}{\longrightarrow} (\hat{Q}', D')} \quad \text{(C\_Rec)}$$

$$\frac{(\hat{Q}, D) \overset{\zeta,\lambda,H}{\longrightarrow} (\hat{Q}', D') \wedge \neg \hat{Q}' \equiv \surd}{(\hat{Q}[> \hat{R}, D) \overset{\zeta,\lambda,H}{\longrightarrow} (\hat{Q}'[> \hat{R}, D')} \quad \text{(C\_Int 1)}$$

$$\frac{(\hat{Q}, D) \overset{\zeta,\lambda,H}{\longrightarrow} (\surd, D')}{(\hat{Q}[> \hat{R}, D) \overset{\zeta,\lambda,H}{\longrightarrow} (\surd, D')} \quad \text{(C\_Int 2)}$$

$$\frac{(\hat{R}, D) \stackrel{\zeta,\lambda,H}{\longrightarrow} (\hat{R}', D')}{(\hat{Q}[> \hat{R}, D) \stackrel{\zeta,\lambda,H}{\longrightarrow} (\hat{R}', D')} \qquad \text{(C\_Int 3)}$$

As an example of the application of these rules, here is a sketch of a derivation of the timed automata (figures 5.2, 5.3) of the flow regulator system of § 5.3.1.

Consider the Flow process term: `rec X.([ReadSensor:85,90]; k!flow.x; (idle [> [PERIOD:10000,10250]); X)`.

Corresponding to the subterm `[ReadSensor:85,90]` is an automaton location $(\texttt{[ReadSensor:85,90]}^h, D)$ where h is an additional clock and D a data environment which supports the numerical variable `x`. According to § 5.4.2.1, this location has invariant $(h \leq 90)$ and by § 5.4.2.2 rule (C_Cmp) there is a transition $(\texttt{[ReadSensor:85,90]}^h, D) \rightarrow (\surd, D^\omega)$ where D^ω is the state of D after performance of the data operation ω. This transition has guard $(h \geq 85)$ and action label ω.

It follows by rule (C_Seq 2) that there is a transition

$$(\texttt{[ReadSensor:85,90];k!flow.x;(idle [> [PERIOD:10000,10250]);X}, D) \stackrel{h>=85,\omega,\{h\}}{\longrightarrow} (\texttt{k!flow.x;(idle [> [PERIOD:10000,10250]);X}, D^\omega) \qquad (5.1)$$

Similarly, a transition by rule (C_Snd), $(\texttt{k!flow.x}, D^\omega) \stackrel{k!\texttt{flow}.v}{\longrightarrow} (\surd, D^\omega)$ (where v is the value of `x` in D^ω) gives, via rule (C_Seq 2) transition

$$(\texttt{k!flow.x;(idle [> [PERIOD:10000,10250]);X}, D^\omega) \stackrel{k!\texttt{flow}.v}{\longrightarrow} (\texttt{(idle [> [PERIOD:10000,10250]);X}, D^\omega) \qquad (5.2)$$

The last location has invariant $(h <= 10250)$.
From $(\texttt{[PERIOD:10000,10250]}, D^\omega) \stackrel{(h>=10000}{\longrightarrow} (\surd, D^\omega)$ one obtains, using rules (C_Int 3) and (C_Seq 2) a transition

$$(\texttt{(idle [> [PERIOD:10000,10250]);X}, D^\omega) \stackrel{h>=10000}{\longrightarrow} (\texttt{X}, D^\omega) \qquad (5.3)$$

Now, let $\hat{Q}$ denote the clocked process term at the source of transition 5.1.

Substituting $recX.\hat{Q}$ for the free process variable X in 5.1-5.3,

$$\begin{aligned}
&(\texttt{[ReadSensor:85,90];k!flow.x;(idle [> [PERIOD:10000,10250])};recX.\hat{Q}, D)\\
&\xrightarrow{h>=85,\omega,\{h\}} (\texttt{k!flow.x;(idle [> [PERIOD:10000,10250])};recX.\hat{Q}, D^{\omega})\\
&\xrightarrow{k!\texttt{flow}.v} (\texttt{(idle [> [PERIOD:10000,10250])};recX.\hat{Q}, D^{\omega})\\
&\xrightarrow{h>=10000} (recX.\hat{Q}, D^{\omega})
\end{aligned} \tag{5.4}$$

The first clocked process term in this sequence is simply $\hat{Q}[recX.\hat{Q}/X]$; so, by rules (C_Rec) and (C_Seq 1) we can infer,

$$\begin{aligned}
&(recX.\hat{Q}, D)\\
&\xrightarrow{h>=85,\omega,\{h\}} (\texttt{k!flow.x;(idle [> [PERIOD:10000,10250])};recX.\hat{Q}, D^{\omega})\\
&\xrightarrow{k!\texttt{flow}.v} (\texttt{(idle [> [PERIOD:10000,10250])};recX.\hat{Q}, D^{\omega}) \xrightarrow{h>=10000} (recX.\hat{Q}, D^{\omega})
\end{aligned}$$

This is the loop comprising figure 5.2 (note that with UPPAAL automata, the data environment D is subsumed into the locations). The locations are given less unwieldy names in the figure, and the action $k!\texttt{flow}.v$ updates variables shared with the channel automaton before synchronising with it. We shall see in the next chapter a refined channel model in which this is accomplished by a single synchronised action.

The automaton of figure 5.3 is derived in a similar way, starting from process term $\texttt{recY.(k?flow.y; [AdjustValve:200,300]; Y)} \equiv recY.\hat{R}$. Rules (C_Rcv) and (C_Cmp) can be used in conjunction with (C_Seq 2) to obtain transitions of the form

$$\begin{aligned}
&(\texttt{k?flow.y;[AdjustValve:200,300]);Y}, D)\\
&\xrightarrow{k?\texttt{flow.v},\{h,h^u\}} (\texttt{[AdjustValve:200,300]);Y}, D[y := v])\\
&\xrightarrow{h>=200,\texttt{Adj}} (\texttt{Y}, D[y := v])
\end{aligned}$$

Recognising that the first clocked process term in the sequence is $\hat{R}$, one

substitutes $recY.\hat{R}$ for Y and obtains, using rule (C_Rec),

$$(recY.\hat{R}, D)$$
$$\xrightarrow{k?\texttt{flow.v},\{h,h^u\}} (\texttt{[AdjustValve:200,300]}); recY.\hat{R}, D[y := v])$$
$$\xrightarrow{h>=200,\texttt{Adj}} (recY.\hat{R}, D[y := v])$$

It will be shown that, under a few simple assumptions, a general bCANdle system (P, N, D) in which $P = P_1|...|P_n$ and the process terms P_i are constructed without $|$, is *strongly equivalent* to the timed transition system of a parallel product (see §2.2.3) of automata including the $\mathbb{A}(\hat{P}_i, D_i)$ constructed as in subsection 5.4.2.

5.4.3 Strong Bisimulation and Strong Equivalence in bCANdle

Given an equivalence relation $\approx$ between states, a *strong $\approx$-bisimulation (up to $\simeq$)* is a relation R between states which is a bisimulation (up to $\simeq$) as above, and also $R \subseteq \approx$ – i.e. $(\sigma_1, \sigma_2) \in R \Rightarrow \sigma_1 \approx \sigma_2$. This notion of strong equivalence may be applied to the timed transition systems of bCANdle and of timed automata. In this case, $\approx$ is what in [79] is termed *context equivalence*: between two bCANdle systems, the network and data model states equate: $(P, N, D) \approx (P', N', D')$ iff $N = N', D = D'$.

Two bCANdle systems are *strongly equivalent*,

$$(P_1, N_1, D_1) \simeq (P_2, N_2, D_2)$$

iff there is a $\approx$-bisimulation between them. From this Kendall develops a notion of strong equivalence of *closed, guarded* process terms. Closed terms are those in which every process variable is bound by the `rec` "quantifier" and guarded process terms are those in which every process variable is guarded in the sense of §4.3.9. Kendall's process term equivalence $P_1 \simeq P_2$ holds iff $(P_1, N, D) \simeq (P_2, N, D)$ for all admissible (in a sense he makes precise) network and data models. This is a notion of *semantic equivalence* of process

terms: it is a congruence with respect to all the process term operators, and $\vdash P_1 = P_2$ implies $P_1 \simeq P_2$. This is the *soundness* of the equational laws referred to at the end of §4.3.9.

As Kendall remarks, semantic equivalence extends in a natural way to *clocked* process terms: $\hat{P}_1 \simeq \hat{P}_2$ iff timed systems initiating from $\hat{P}_1, \hat{P}_2$ are strongly $\approx$-bisimilar. In the timed-automaton formalism presented in this chapter, this means that whenever an automaton $\mathbb{A}(\hat{P}_1, D)$ is composed with a suitable (set of) network channel automata, the timed transition system of the product is $\approx$-bisimilar to the timed transition system of the product of $\mathbb{A}(\hat{P}_2, D)$ with the same network. The equational laws are *sound* in the sense that any equation $\hat{P}_1 = \hat{P}_2$ derived from them implies $\hat{P}_1 \simeq \hat{P}_2$.

5.5 bCANdle as a Parallel Product

5.5.1 Theorem

Suppose a bCANdle system (P, N, D) in which $P = P_1|...|P_n$ and the process terms P_i are constructed without $|$. Suppose further that the variables occurring in P_i are disjoint from the variables occurring in P_j when $i \neq j$. Let D_i denote the data environment which results from restricting D to the variables occurring in P_i, and $\hat{P}_i$ the *clocked* process term resulting from P_i according to § 5.4.1. Suppose the time constraints on atomic operations are natural-number-valued and that clocks are allocated safely and with static control as explained at the beginning of § 5.4.

Let $N_k, k \in K$ denote the timed automata representing the CAN channels as described in § 5.3.

Then the timed transition system of the product $\|_{i=1}^{n}\ \mathbb{A}(\hat{P}_i, D_i)\ \|\ (\|_{k\in K}\ N_k)$ is strongly equivalent to that of (P, N, D).

5.5.2 Construction and Proof

The proof of this theorem is achieved by constructing a strong $\approx$-bisimulation where $\approx$ is context equivalence suitably extended to include timed states of

of product automaton. The construction is described here, and the bulk of the details of the proof appear in appendix B.

Suppose N consists of CAN channels $N_k, k \in K$. A channel N_k will be in one of the states $(\downarrow, u)^h$, $(\overset{t_1,t_2}{\rightsquigarrow} m, u)^h$, $(\uparrow m; u)^h$, $(m \overset{t_1,t_2}{\rightsquigarrow}, u)^h$ explained in § 4.2. These map into the locations `Free`, `PreAcc`, `Acc`, `PostAcc` of the timed-automaton model of the CAN channel displayed in § 5.3. Let $C_k (k \in K)$ denote an instance of this automaton, corresponding to N_k.

The product automation proposed to represent the bCANdle system is $(\|_{i=1}^{n} \mathbb{A}(\hat{P}_i, D_i)) \parallel (\|_{k \in K} C_k)$.

The main task is to construct a bisimulation relating (P, N, D) to the timed transition system of this product. This will be done by adapting a construction of Kendall's ([79], appendix B).

A general state of the timed transition system of the product automaton has the form

$$((\hat{Q}_i, D_{Qi})_{i=1\ldots m}, (\eta_k)_{k \in K}, v)$$

where $m \leq n$ and v is a valuation of clocks. The remaining parts comprise a vector of $\mathbb{A}(P_i, D_i)$ locations and channel automaton locations.

The bisimulation is constructed via a mapping of such states to bCANdle systems, using an adaptation of the `age` function of [79]. This function is defined on clocked process terms as in [79], definition B2, but without the |-clause. Its definition is as below.

$\texttt{unclk}(\hat{Q})$ is the process term obtained by removing the clock references from the clocked process term $\hat{Q}$: $\texttt{unclk}(k!i.x) \triangleq k!i.x$; $\texttt{unclk}(k?i.x) \triangleq k?i.x$; $\texttt{unclk}([\omega, t_1, t_2]^h) \triangleq [\omega, t_1, t_2]$; $\texttt{unclk}(\gamma \rightarrow \hat{Q}_1) \triangleq \gamma \rightarrow \texttt{unclk}(\hat{Q}_1)$; $\texttt{unclk}(\hat{Q}_1; \hat{Q}_2) \triangleq \texttt{unclk}(\hat{Q}_1); \texttt{unclk}(\hat{Q}_2)$ and similarly for $\texttt{unclk}(\hat{Q}_1 + \hat{Q}_2)$, $\texttt{unclk}(\hat{Q}_1 [> \hat{Q}_2)$; $\texttt{unclk}(recX.\hat{Q}_1) \triangleq recX.\texttt{unclk}(\hat{Q}_1)$; $\texttt{unclk}(X) \triangleq X$.

$t \dot{-} t'$ is the greater of $t - t', 0$.

$$\begin{aligned}
\mathtt{age}(k!i.x, v) &\triangleq k!i.x \\
\mathtt{age}(k?i.x, v) &\triangleq k?i.x \\
\mathtt{age}([\omega, t_1, t_2]^h, v) &\triangleq [\omega, t_1', t_2'] \text{ where } t_i' = t_i \dot{-} v(h) \\
\mathtt{age}(\gamma \to \hat{Q}, v) &\triangleq \mathtt{unclk}(\hat{Q}) \\
\mathtt{age}(\hat{Q}; \hat{Q}', v) &\triangleq \mathtt{age}(\hat{Q}, v); \mathtt{unclk}(\hat{Q}') \\
\mathtt{age}(\hat{Q} + \hat{Q}', v) &\triangleq \mathtt{age}(\hat{Q}, v) + \mathtt{age}(\hat{Q}', v) \\
\mathtt{age}(\hat{Q}[> \hat{Q}', v) &\triangleq \mathtt{age}(\hat{Q}, v)[> \mathtt{age}(\hat{Q}', v) \\
\mathtt{age}(recX.\hat{Q}, v) &\triangleq \mathtt{age}(\hat{Q}[recX.\hat{Q}/X], v)
\end{aligned}$$

Similarly, a function `age` maps a timed state of a channel automaton to a bCANdle network channel state thus.

$$\begin{aligned}
\mathtt{age}(\mathtt{Free}, v) &\triangleq (\downarrow, u) \\
\mathtt{age}(\mathtt{PreAcc}, v) &\triangleq (\overset{t_1,t_2}{\rightsquigarrow} m, u) \text{ where } t_1 = l \dot{-} v(h), t_2 = u \dot{-} v(h) \\
\mathtt{age}(\mathtt{Acc}, v) &\triangleq (\uparrow m, u) \\
\mathtt{age}(\mathtt{PostAcc}, v) &\triangleq (m \overset{t_1,t_2}{\rightsquigarrow}, u) \text{ where } t_1 = L \dot{-} v(h), t_2 = U \dot{-} v(h)
\end{aligned}$$

`Free`, `PreAcc`, `Acc`, `PostAcc` are the locations of a channel automaton in which m is the current message and u the current state of the message queue for the channel, h a clock in the channel automaton reset when it goes to `PreAcc` or `PostAcc`. The notation on the right hand side is that of section 4.2 for a bCANdle network channel.

A general state of the timed transition system of the product automaton, maps to a bCANdle system:

$$((\hat{Q}_i, D_{Qi})_{i=1\ldots n}, (\eta_k)_{k\in K}, v) \longmapsto (|_{i=1}^n \mathtt{age}(\hat{Q}_i, v), (\mathtt{age}(\eta_k, v))_{k\in K}, \sqcup_{i=1}^n D_{Qi}) \tag{5.5}$$

– the data environment component $D_{Q1} \sqcup \ldots \sqcup D_{Qn}$ is formed in the natural way from the data environments D_{Qi} over disjoint sets of variables.

To complete the proof it suffices to show that the graph B of this mapping

is a strong $\approx$-bisimulation up strong equivalence ($\simeq$) between the two timed transition systems, the transition graph of the product automaton, and the transition graph of of the bCANdle system, where $\approx$ is a relation of context-equivalence suitably generalised to states of either timed transition systems.

Let σ_1, σ_2 denote two states of either timed transition system; say $\sigma_1 \approx \sigma_2$ (σ_1 is *context equivalent* to σ_2) iff σ_1, σ_2 fit one of the following clauses:

- Two bCANdle systems: $(Q, N, D) \approx (Q', N', D')$ iff $N = N', D = D'$
- A bCANdle system and an automaton state:
  $((\hat{Q}_i, D_{Qi})_{i=1...m}, (\eta_k)_{k\in K}, v) \approx (Q, N, D)$ iff
  $(Q, N, D) \approx ((\hat{Q}_i, D_{Qi})_{i=1...m}, (\eta_k)_{k\in K}, v)$ iff
  $(\forall k \in K) N_k = \mathtt{age}(\eta_k, v)$ and $D = D_{Q1} \sqcup ... \sqcup D_{Qm}$
- Two automaton states:
  $((\hat{Q}_i, D_{Qi})_{i=1...m}, (\eta_k)_{k\in K}, v) \approx ((\hat{Q}'_i, D'_{Qi})_{i=1...m'}, (\eta'_k)_{k\in K}, v')$ iff
  $(\forall k \in K)\mathtt{age}(\eta_k, v) = \mathtt{age}(\eta'_k, v')$ and $D_{Q1} \sqcup ... \sqcup D_{Qm} = D'_{Q1} \sqcup ... \sqcup D'_{Qm'}$

To see that the mapping (5.5) provides a strong $\approx$-bisimulation up to strong equivalence ($\simeq$) one needs to check, for $((\hat{Q}_i, D_{Qi})_{i=1...m}, (\eta_k)_{k\in K}, v)$ as a general timed state of the automaton, and $(|_{i=1}^n \mathtt{age}(\hat{Q}_i, v), (\mathtt{age}(\eta_k, v))_{k\in K}, \sqcup_{i=1}^n D_{Qi})$ as the bCANdle system to which it maps,

1. $((\hat{Q}_i, D_{Qi})_{i=1...m}, (\eta_k)_{k\in K}, v) \approx (|_{i=1}^n \mathtt{age}(\hat{Q}_i, v), (\mathtt{age}(\eta_k, v))_{k\in K}, \sqcup_{i=1}^n D_{Qi})$;
2. For every transition λ from $(|_{i=1}^n \mathtt{age}(\hat{Q}_i, v), (\mathtt{age}(\eta_k, v))_{k\in K}, \sqcup_{i=1}^n D_{Qi})$ in the bCANdle timed transition system there is a transition with the same label λ from $((\hat{Q}_i, D_{Qi})_{i=1...m}, (\eta_k)_{k\in K}, v)$, and the targets are context equivalent to states related by the mapping;
3. For every transition λ from $((\hat{Q}_i, D_{Qi})_{i=1...m}, (\eta_k)_{k\in K}, v)$ there is a transition with the same label λ from $(|_{i=1}^n \mathtt{age}(\hat{Q}_i, v), (\mathtt{age}(\eta_k, v))_{k\in K}, \sqcup_{i=1}^n D_{Qi})$, and the targets are context equivalent to states related by the mapping.

The first of these requirements is met by definition of context equivalence.

The second is established by considering a number of cases and using induction on the computation of $|_{i=1}^{n}\mathtt{age}(\hat{Q}_i, v)$. Details appear in appendix B.

The third is established in a way symmetrical with the second, again by induction on the computation of $|_{i=1}^{n}\mathtt{age}(\hat{Q}_i, v)$. Details are in appendix B.

5.6 Conclusion

In this chapter we have established a formal construction which derives from a bCANdle model of a broadcasting embedded system a semantically equivalent parallel composite of timed automata. This will be useful in the sequel because it will allow one to reason *compositionally* about bCANdle models. The components of a model can be examined individually, replaced with abstractions and reasoned about; then, using suitable compositionality theorems developed in chapter 7, properties of the model can be inferred from properties of a model in which come components have been abstracted. This, as we shall see, is a useful thing to be able to do in situations where the model's state space is "blowing up", becoming unmanagably large.

Chapter 6

Composition without Shared Variables

Chapter 5 showed how a CAN-based distributed (broadcasting) embedded system could be modelled by a parallel composition of timed automata, and showed it strongly equivalent to a bCANdle system. This approach depends on the construction of a CAN channel as a *process* or timed automaton and indeed such a construction arose in a natural way from the definition of the behaviour of a CAN channel. The aim is to use this *compositional* method of modelling in order to verify properties of CAN systems by inference from properties of components, or of system models in which components have been abstracted in some way. The compositionality theorems presented in chapter 7 provide a theoretical basis for doing this, using a form of *assume-guarantee* reasoning; but this requires the composition of timed automata to have *no shared variables*. The naturally arising CAN automaton described above uses shared variables. A formally equivalent structure exists which looks much less natural and elegant, but does the job *without* shared variables. The present chapter presents such a structure and shows it to be equivalent to the formalism already developed.

6.1 Another CAN Channel Automaton

An alternative formulation of a timed automaton for a CAN channel is introduced and will be shown to be strongly equivalent to that of chapter 5.

This automaton is based closely on that of §5.3: indeed, it is like its prototype, an UPPAAL process template, and has the same locations and the same cyclic behaviour.

The major difference is motivated by the need to do without shared variables. The original model (§5.3) shared with its environment

- Variables `iIn`, `vIn` of the types of the i-values and the v-values respectively. These hold the type and the value of a CAN message $i.v$ being put onto the network.
- Variables `iDlvr`, `vDlvr` of the types of the i-values and the v-values respectively. These hold the type and the value of a CAN message $i.v$ being delivered from the network to a receiving process.
- The channel automaton receives a synchronisation on an UPPAAL "urgent channel" `nMsg?` when a process wishes to send a message: it reacts by copying the message payload in `vIn` into `Q[iIn]`.
- When a message in the channel is ready to be handed off to the receiving process, its type and payload are copied to `iDlvr` and `vDlvr` and a broadcast synchronisation issued on an UPPAAL 'broadcast channel" `dMsg!`.

Figure 6.1 shows the new channel automaton in an abbreviated form. The logical flow is for the most part the same as that of the original §5.3 channel: compare this diagram with figure 5.1 on page 93. (Note: the legend (c) denotes a committed location in the UPPAAL sense.)

The array `Q` is carried over from the old model. This is indexed by *message identifiers*, the possible values of component i of a CAN message. The type of these is a subrange $0 \,..\, (m-1)$ where m is the number of message types supported by the channel; and the type of the array entries is that of the message payload, the component v of a CAN message. For the purposes

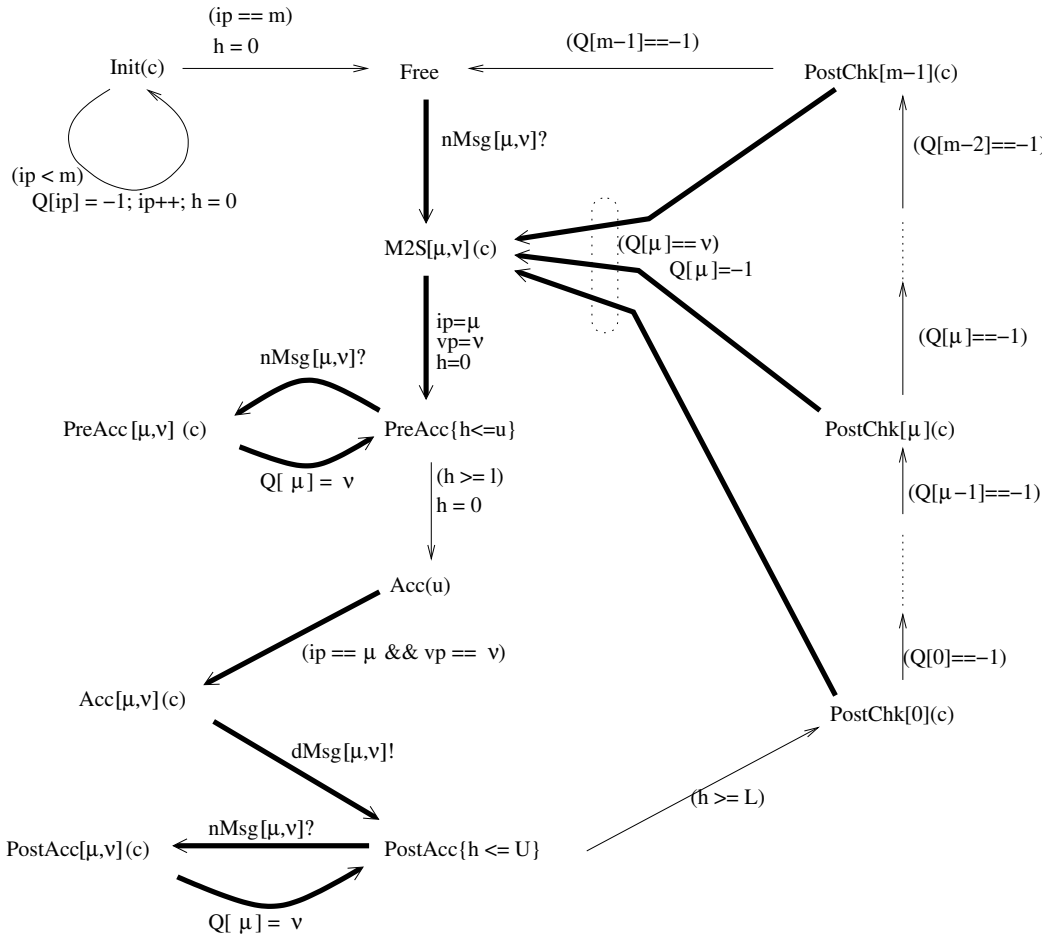


Figure 6.1: Automaton to model a CAN Channel without shared variables

of the present work, this is also a subrange of integers from 0 to some maximum supported v value. `Q[i]` may also hold the value `FREE` (actually, -1) indicating that this message type is currently vacant.

The important differences in the new model are as follows. Suppose there are m different message types and n different values (that is, a message type is an integer in the range 0 .. $(m-1)$ and a message value is an integer in the range 0 .. $(n-1)$).

- In place of `iIn`, `vIn` and the single urgent channel `nMsg` are $m \times n$ urgent channels `nMsg`$[\mu, \nu]$, $0 \leq \mu < m,\ 0 \leq \nu < n$.

- In place of the committed location `M2S` is the family of committed locations $\texttt{M2S}[\mu,\nu]$, $0 \leq \mu < m,\ 0 \leq \nu < n$, with urgent synchronising edges decorated $\texttt{nMsg}[\mu,\nu]\texttt{?}$ fanning out from location `Free` to $\texttt{M2S}[\mu,\nu]$. In figure 6.1 the thick arrows denote such *bundles* of edges. These edges play a role analogous to the synchronising edge

$$\texttt{Free} \xrightarrow{\texttt{nMsg?,Q[iIn]:=vIn,...}} \texttt{M2S}$$

  in figure 5.1 (p93).

- There are also families of committed locations $\texttt{PreAcc}[\mu,\nu]$, $\texttt{Acc}[\mu,\nu]$, $\texttt{PostAcc}[\mu,\nu]$, $0 \leq \mu < m,\ 0 \leq \nu < n$ in addition to locations `PreAcc`, `Acc` (urgent), `PostAcc`. The thick arrows in the figure between these locations denote bundles of edges fanning-in or fanning-out.

- Note, in particular, the arrows $\texttt{Acc}[\mu,\nu] \xrightarrow{\texttt{dMsg}[\mu,\nu]\texttt{!}} \texttt{PostAcc}$ fanning in. These are analogous to the edges in figure 5.1 (p93),

$$\texttt{Acc} \xrightarrow{\texttt{iDlvr := ip,vDlvr := vp}} \texttt{S1} \xrightarrow{\texttt{dMsg!}} \texttt{PostAcc}$$

- Similarly, the edges labelled $\texttt{nMsg}[\mu,\nu]\texttt{?}$ from `PreAcc` to $\texttt{PreAcc}[\mu,\nu]$ and from `PostAcc` to $\texttt{PostAcc}[\mu,\nu]$, followed by edges back again guarded by the condition $(Q[\mu] == \nu)$, are analogous to the self-loops on `PreAcc` and `PostAcc` in figure 5.1.

- The array `Q[]` is still present in the new model but is a local variable. The shared variables `iIn`, `vIn`, `iDlvr`, `vDlvr` have been replaced by the multiple synchronisations $\texttt{nMsg}[\mu,\nu]$, $\texttt{dMsg}[\mu,\nu]$.

- Note the edges from `PostAcc` to `PostChk`[0] to ... to `PostChk`$[\mu]$ ... to `PostChk`$[m-1]$ to `Free` with bundles of edges, an edge from `PostChk`$[\mu]$ to $\texttt{M2S}[\mu,\nu]$ guarded by $(Q[\mu] == \nu)$ and with update $Q[\mu] = -1$. These provide for messages being removed from the queue in priority order ($\mu == 0$ first) and sent.

The corresponding part of the old model (fig 5.1, p93) is the guarded self-loop at location `M2S` together with the guarded edges from here to locations `Free` and `PreAcc`, using local variables `ip, vp`. In the new model, the multiple paths indexed by μ and ν values replace these local variables.

It should be borne in mind throughout this account that the μ, ν are *metavariables* – every occurrence of them in the foregoing in reality occurs as specific values occurring in the context of an indexed set of UPPAAL tokens – locations, synchronisation channels and so forth.

The following pseudo code is an UPPAAL `xta` listing of the new channel: compare this with the listing in §5.3.2. The metavariables μ, ν in the account above appear here as μ, ν also; lines containing these metavariables and decorated with inequalities on them, such as $[0 \leq \mu < m]$ and/or $[0 \leq \nu < n]$ are to be understood as being repeated over all (integer) values of μ, ν satisfying the inequalities. Symbols rendered in figure 6.1 and the commentary on it, as for instance `M2S_`$[\mu, \nu]$, are written below as `M2S_`μ`_`ν and so forth, and expanded into "proper" UPPAAL `xta` as `M2S_000_000`, `M2S_000_001`, ..., `M2S_001_000`, ... and so on.

```
 process Channel(const int l, const int u, const int L, const int U,
  urgent chan  nMsg_μ_ν[0 ≤ μ < m, 0 ≤ ν < n],
  broadcast chan dMsg_μ_ν[0 ≤ μ < m, 0 ≤ ν < n]) {
clock h;
int[0,m] ip;
int[0,n − 1] vp;
int[-1,n − 1] Q[m];

state
  Init, Free,
  M2S_μ_ν[0 ≤ μ < m, 0 ≤ ν < n], PreAcc_μ_ν[0 ≤ μ < m, 0 ≤ ν < n],
  Acc_μ_ν[0 ≤ μ < m, 0 ≤ ν < n], PostAcc_μ_ν[0 ≤ μ < m, 0 ≤ ν < n],
  PreAcc{h<=u}, Acc, PostAcc{h<=U},
  PostChk_μ[0 ≤ μ < m];
```

```
  commit
    Init,
    M2S_μ_ν[0 ≤ μ < m, 0 ≤ ν < n], PreAcc_μ_ν[0 ≤ μ < m, 0 ≤ ν < n],
    Acc_μ_ν[0 ≤ μ < m, 0 ≤ ν < n],
    PostAcc_μ_ν[0 ≤ μ < m, 0 ≤ ν < n],
    PostChk_μ[0 ≤ μ < m];

  urgent
    Acc;
  init
    Init;

trans
  Init -> Init {
    guard ip < μ;
    assign Q[ip] = -1, ip++, h = 0;
  },
  Init -> Free { guard ip == μ; assign h = 0; },
  Free -> M2S_μ_ν { sync nMsg_μ_ν?; }[0 ≤ μ < m, 0 ≤ ν < n],
  M2S_μ_ν -> PreAcc { assign ip=μ, vp=ν, h = 0; }
                                        [0 ≤ μ < m, 0 ≤ ν < n],
  PreAcc -> PreAcc_μ_ν { sync nMsg_μ_ν?; }[0 ≤ μ < m, 0 ≤ ν < n],
  PreAcc_μ_ν -> PreAcc { assign Q[μ] = ν;}[0 ≤ μ < m, 0 ≤ ν < n],
  PreAcc -> Acc { guard h >= l;  assign h = 0; },
  Acc -> Acc_μ_ν { guard ip == μ && vp == ν; }
                                        [0 ≤ μ < m, 0 ≤ ν < n],
  Acc_μ_ν -> PostAcc {sync dMsg_μ_ν!; }[0 ≤ μ < m, 0 ≤ ν < n],
  PostAcc -> PostAcc_μ_ν {sync nMsg_μ_ν?;}[0 ≤ μ < m, 0 ≤ ν < n],
  PostAcc_μ_ν -> PostAcc {assign Q[μ]=ν;}[0 ≤ μ < m, 0 ≤ ν < n],

  PostAcc -> PostChk_0 { guard h >= L; },
  PostChk_(μ − 1)->PostChk_μ {guard Q[(μ − 1)] == -1;}[1 ≤ μ < m],
  PostChk_(m − 1)->Free {guard Q[(m − 1)] == -1;},
  PostChk_μ -> M2S_μ_ν {guard Q[μ] == ν; assign Q[μ] = -1;}
                                        [0 ≤ μ < m, 0 ≤ ν < n];
```

This pseudo-UPPAAL code is expanded, for a particular choice of value for the parameters m and n, the number of message types and or message payload values supported, by a software tool `CANGen` devised by the author. The `CANGen` output for $m = 3$, $n = 4$ corresponding to the pseudocode above

appears as appendix C.

It will be evident that the no-shared-variables model of the CAN channel has a large number – $m \times n$ – of locations and synchronisation channels. In effect these have been taken on in lieu of the shared variables $(iIn, vIn, iDlvr, iDlvr)$ of the old model, whose values occupied a similarly sized portion of the state space. Technically, this change is not expensive in terms of the state space the model-checker must search, although admittedly it is conceptually unwieldy. It will transpire, however, that even with the large number of locations and synchronisations, the compositional approach afforded by this model does afford considerable speed-up and economy in model-checking.

As to the unwieldiness of the large number of locations and synchronisations: in the sequel and in practice we spend very little time looking at the "full" automaton, the `CANGen` output. the pseudocode is easier to read and also to diagram. As will be seen in chapter 8 one can concisely draw automaton diagrams corresponding to the pseudocode, in which a single edge or location may represent a "bundle" of edges or locations indexed by μ, ν (which also figure in guards, synchronisations, updates decorating the edges). It is thus possible to retain a simple view of the model in spite of the potentially large number of locations and edges.

Practical limits to the size of $m \times n$ will be discussed in chapter 8. Of course, in practice it may well be possible simplify a model checking problem by working with an abstraction in which these parameters are of modest size.

6.2 The Equivalence of the Two CAN Models

The new model was developed with reference to the old one, and intended to have exactly the same semantics, so that the equivalence with bCANdle is preserved. To demonstrate this equivalence formally a bisimulation will be developed between the following two systems -

1. An UPPAAL system combining an old-style CAN channel model with a sending process which repeatedly sends messages of some type μ $(0 \leq$

$\mu < m$) and value ν ($0 \leq \nu < n$), chosen non-deterministically, and a receiving process which indefinitely and repeatedly waits for messages from the channel and takes receipt of them.

The sender's duty cycle will set shared variable `iIn` to μ and `vIn` to ν before offering a synchronisation on `nMsg`. The receiver will, on each synchronisation with `dMsg` save the shared variables `iDlvr` and `vDlvr` and switch to a location determined by their values. See figure 6.2. As before, (c) denotes an UPPAAL *committed* location and (u) an *urgent* location; and the thick arrows a bundle of edges fanning in or out, replicated over all μ, ν.

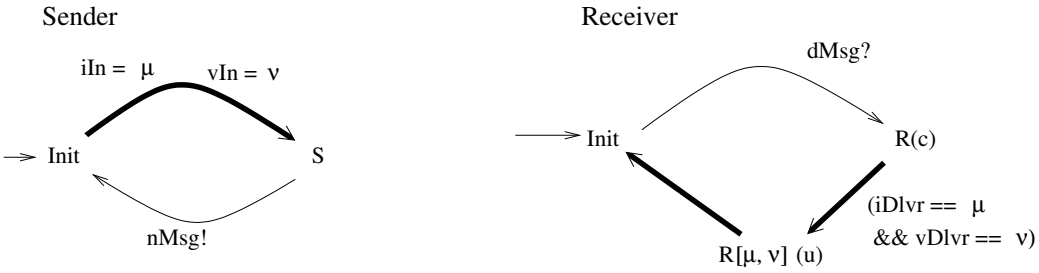


Figure 6.2: A Sender and a Receiver process to drive the old Channel automaton

2. An UPPAAL system combining a new-style CAN channel model with a sending process which repeatedly sends messages of some type μ and value ν, chosen non-deterministically, and a receiving process which indefinitely and repeatedly waits for messages from the channel and takes receipt of them – just as above.

   In this case the sender will non-deterministically choose a synchronization `nMsg_`μ`_`ν to offer to the channel. The receiver will wait for any one of the $m \times n$ synchronisations `dMsg_`μ`_`ν. See figure 6.3.

The bisimulation will be a relation between states of the two systems.

The variables of the "new" system (sender, new channel model, receiver) are just the local variables of the channel automaton: integer subranges `ip`,

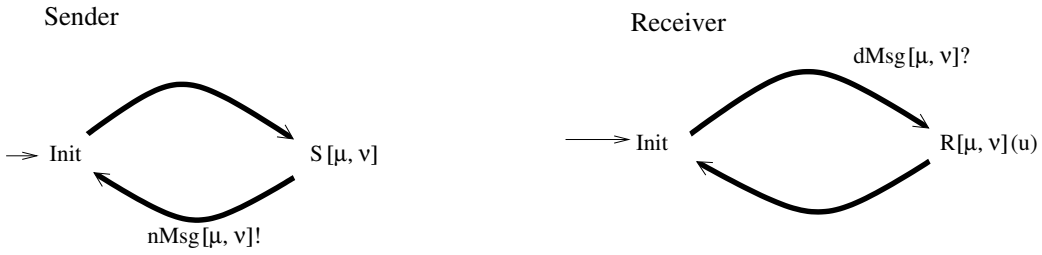


Figure 6.3: A Sender and a Receiver process to drive the new Channel automaton

`vp`, array `Q[]`, clock `h`. Thus a state of the "new" system is a valuation of these variables together with a vector of locations.

The "old" system (sender, old channel model, receiver) has just these variables plus integer subranges `iIn`, `vIn`, `iDlvr`, `vDlvr`. Thus a state is a valuation of these variables together with a vector of locations.

The semantics of UPPAAL committed locations say that no time may pass in a state containing a committed location, and that after a transition to such a state, the next transition must be from a state containing a committed location. This means, that in the absence of simultaneous committed states elsewhere in the model a sequence of transitions

$$C_1 \rightarrow C_2 \rightarrow ... \rightarrow C_n \rightarrow D$$

in which all but the last location are committed may be regarded as atomic. The committed locations are interposed simply to enforce a particular sequence of actions – synchronisations, assignments, including choices driven by guards. This, indeed, is how committed locations have been used in the present work. Let us call a sequence of edges like this a *committed sequence*.

6.2.1 Definition - Weak Bisimulation of UPPAAL Systems

Bearing this in mind, a bisimulation between the "old" and the "new" system will be developed as a relation between states of the two systems in which

only non-committed locations figure, and such that whenever we have related states $\sigma_{old}, \sigma_{new}$,

1. For any sequence of timed transitions

$$\sigma_{old} \to \sigma^1_{old} \to \sigma^2_{old} \to ... \to \tau_{old}$$

in which the transitions arise from a committed sequence of edges of one of the components, there is a sequence (based on a committed sequence in one of the components)

$$\sigma_{new} \to \sigma^1_{new} \to \sigma^2_{new} \to ... \to \tau_{new}$$

for some τ_{new} related to τ_{old};

2. For any sequence of timed transitions

$$\sigma_{new} \to \sigma^1_{new} \to \sigma^2_{new} \to ... \to \tau_{new}$$

in which the transitions arise from a committed sequence of edges of one of the components, there is a sequence (based on a committed sequence in one of the components)

$$\sigma_{old} \to \sigma^1_{old} \to \sigma^2_{old} \to ... \to \tau_{old}$$

and τ_{old} and τ_{new} are related.

Any such relation we term a *weak bisimulation* of UPPAAL systems. Of course, it is strong enough for us to regard the systems as equivalent in their external behaviour.

6.2.2 The bisimulation

Let us write (S_o, C_o, R_o, v_o) for a state of the "old" system: these are, respectively, *non-committed* locations of the Sender (figure 6.2), the Channel automaton (figure 5.1, page 93), and the Receiver (figure 6.2); and a valuation of the clock and other variables of the old system.

Similarly we write (S_n, C_n, R_n, v_n) for a state of the "new" system (figures 6.3, 6.1).

Note that the domain of v_o is the domain of v_n together with the shared variables `iIn`, `vIn`, `iDlvr`, `vDlvr`, and $C_o, C_n \in \{\texttt{Free}, \texttt{PreAcc}, \texttt{Acc}, \texttt{PostAcc}\}$.

Let us say $(S_o, C_o, R_o, v_o)B(S_n, C_n, R_n, v_n)$ iff -

1. $C_o = C_n \in \{\texttt{Free}, \texttt{PreAcc}, \texttt{Acc}, \texttt{PostAcc}\}$;
2. v_o and v_n agree on the channel local variables;
3. $S_o = \texttt{Init} \Leftrightarrow S_n = \texttt{Init}$;
4. $\forall\mu\forall\nu : (S_o = \texttt{S} \wedge v_o(\texttt{iIn}) = \mu \wedge v_o(\texttt{vIn}) = \nu) \Leftrightarrow S_n = \texttt{S}[\mu, \nu]$;
   (the quantifiers range over $0 \leq \mu < m$, $0 \leq \nu < n$)
5. $R_o = \texttt{Init} \Leftrightarrow R_n = \texttt{Init}$;
6. $\forall\mu\forall\nu : (R_o = \texttt{R}[\mu, \nu] \Leftrightarrow R_n = \texttt{R}[\mu, \nu]$;
7. $\forall\mu\forall\nu : (R_o = \texttt{R}[\mu, \nu] \Rightarrow v_0(\texttt{iDlvr}) = \mu \wedge v_0(\texttt{vDlvr}) = \nu$.

6.2.3 Theorem

The relation between state of the "old" and the "new" systems defined by §6.2.2 is a weak bisimulation of UPPAAL systems.

Proof. It will be useful to refer to figures 5.1 (page 93) and 6.1 (page 117) and the (pseudo) UPPAAL code relating to these diagrams.

In both systems, any run begins from a state in which all variables have value 0, and both channel models pass initially through committed locations to `Free`, where the message queue `Q[]` has been initialised so that all entries are $-1 =$ `FREE`. The two `FREE` locations are related.

A committed sequence in the "old" model from `Free` to `PreAcc` will necessarily involve first synchronising on `nMsg` with the sender process (`ip`, `vp` will be set to the type and payload of the message sent), then the channel's local clock h is reset, then (while $l \leq h \leq u$) there may be 0 or more transitions on the loop at `PreAcc` synchronising on `nMsg` with the sender process and

resulting in the queue being populated with more messages. The corresponding sequence of events in the "new" model is first, a transition synchronising with the sender process on $\texttt{nMsg}[\mu, \nu]$ (for some μ, ν), then the message type and payload value (in fact, μ, ν) are saved to `ip`, `vp` and the channel's local clock h reset, then (while $l \leq h \leq u$) there are 0 or more transition sequences from `PreAcc` via a synchronisation with the sender on *some* $\texttt{nMsg}[\mu, \nu]$ (in general different from the previous μ, ν) to committed location $\texttt{PreAcc}[\mu, \nu]$ and back via a transition which updates $Q[\mu]$ to ν. Thus, the variables `h`, `ip`, `vp`, `Q[]` end up in the same state in the two models. From this one can find, for any sequence in the old model $(..., \texttt{Free}, ...) \rightarrow ... \rightarrow (..., \texttt{PreAcc}, ...)$, a corresponding sequence in the new model leading from a *related* source state to a *related* target state.

The reasoning is simpler in the case of a committed sequence in either model from `PreAcc` to `Acc` as the two channel models correspond closely. The sequence in the old model will begin with 0 or more transitions on the loop at `PreAcc` synchronising on `nMsg` with the sender process and resulting in the queue being populated with more messages. The corresponding sequence in the new model will begin with 0 or more transition sequences from `PreAcc` via a synchronisation with the sender on *some* $\texttt{nMsg}[\mu, \nu]$ to $\texttt{PreAcc}[\mu, \nu]$ and back via a transition which updates $Q[\mu]$ to ν. As in the previous case, the two corresponding sequences update the system state in the same way and the bisimilarity of sequences $(..., \texttt{PreAcc}, ...) \rightarrow ... \rightarrow (..., \texttt{Acc}, ...)$ is preserved.

A committed sequence in the old model from `Acc` to `PostAcc` updates shared variables `iDlvr`, `vDlvr` from `ip`, `vp`, then synchronises on `dMsg` with the receiver process, then possibly (while $L \leq h \leq U$) does 0 or more transitions on the loop at `PostAcc` synchronising on `nMsg` with the sender process and resulting in the queue being populated with more messages. A corresponding sequence in the new model goes from `Acc` to $\texttt{Acc}[\mu, \nu]$ matching μ, ν to the current values of `ip`, `vp`, then synchronising with the receiver process on $\texttt{dMsg}[\mu, \nu]$. This may be followed (while $L \leq h \leq U$) with 0 or more transition sequences from `PostAcc` via a synchronisation with the sender on *some* $\texttt{nMsg}[\mu, \nu]$ to committed location $\texttt{PostAcc}[\mu, \nu]$ and back via a transition which updates $Q[\mu]$ to ν. Again it can be seen that correspond-

ing sequence in the two models update state in equivalent ways, so that bisimilarity of sequences of the form $(..., \texttt{Acc}, ...) \rightarrow ... \rightarrow (..., \texttt{PostAcc}, ...)$ is preserved.

A committed sequence in the two models from `PostAcc` will begin with 0 or more sub-sequences synchronising with the sender `nMsg` or $\texttt{nMsg}[\mu, \nu]$ as the case may be and, as we have seen, updating the message queue in equivalent ways. After this, the sequence in the old model goes to committed location `PostChk` resetting `ip` to 0, then executes repeated `ip++` updates until *either*

- a message `Q[ip]` is found waiting to be sent:v there is a move to committed location `M2S` then to `PreAcc` and `vp` is set to `Q[ip]`, `Q[ip]` to `FREE` an `h` to 0; *or*

- `ip` reached the end of the queue (`QCAP` or m in the new model) finding no waiting messages: in this case the sequence moves to location `Free`.

An examination of the committed locations $\texttt{PostChk}[\mu]$, $0 \leq \mu < m$ and the guarded edges between them and from them to $\texttt{M2S}[\mu, \nu]$ will show exactly the same logic. If μ is the smallest such that $Q[\mu]$ is not -1 (`FREE`) the sequence goes $\texttt{PostAcc} \rightarrow \texttt{PostChk[0]} \rightarrow ... \rightarrow \texttt{M2S}[\mu, \nu]$ (for some ν) and thence to `PreAcc`, updating `ip`, `vp` appropriately and resetting the clock. If the queue is empty, the sequence rather goes all the way to $\texttt{PostChk}[m-1]$ and thence to `Free`. (m is the number of distinct message types supported – `QCAP` in the old model.)

Thus, again, the sequences do equivalent things to the system state, and the bisimilarity is preserved.

6.3 Conclusion

The formal derivation from a bCANdle model of a broadcasting embedded system of a semantically equivalent parallel composite of timed automata has now been developed into a construction of a parallel composite of UPPAAL timed automata which communicate without recourse to shared variables. As pointed out at the beginning of the chapter this will allow the construction of

compositional models of CAN-based systems to which assume-guarantee-type compositionality theorems can be applied, to infer properties of a model from properties of abstractions of it. The compositionality results are presented in the next chapter.

Chapter 7

Compositionality Theorems

In this chapter are developed compositionality theorems which will underpin the *compositional* analysis of the broadcasting embedded system models developed in the framework outlined in the chapters above. The definitions and results presented here are motivated by the work of Kaynar, Lynch *et al.* [77], [78] in which they develop a type of timed-automaton-style modelling to which they apply a type of assume-guarantee reasoning. It will be seen that our timed automata and also UPPAAL models map into their formalism, although we have found it instructive to prove analogues of their assume-guarantee result directly for our timed automata and UPPAAL models.

The main result of this work appears in section 7.5, where an assume-guarantee theorem for UPPAAL systems (theorem 7.5.2) is developed and presented. This (and some lesser results in §7.6) underpin the practical work discussed in chapter 8.

7.1 Preamble

The starting point of the present work was the modelling real-time systems with *timed automata* of the type developed by Alur and Dill (see [7]) and with *hybrid automata*, using constructs which generalise them in a natural way. The *UPPAAL* tool set uses essentially hybrid automata of this form also [109].

Kaynar, Lynch, Segala and Vaandrager in [77], [78] develop a form of Timed Automaton for timed systems with data input and output. Their formal definition is different from ours. However they explain how their construct subsumes ours (which they call Alur-Dill automata) and they state (but do not prove in detail) a useful "assume-guarantee" theorem allowing properties of an automaton which is a parallel composition to be inferred from properties of its components.

The purpose of this chapter is to describe some work done comparing the automaton constructions of Kaynar, Lynch *et al.* with ours, and exploring their compositionality theorem, proving a version of it for "our" (Alur/Dill/UPPAAL) type of timed automata.

7.2 Timed Automata

We have been modelling with a notion of timed automata which Kaynar, Lynch *et al.* term "Alur-Dill" automata: $(L, l^0, A, \mathcal{H}, \rightarrow, I)$ where

- L is a set of *control locations*
- l^0 is the *initial location*
- A is a set of *discrete action labels*
- $\mathcal{H}$ is a set of *clock variables*
- $\rightarrow$ is a *transition relation*. Specifically, let $\mathcal{Z}(\mathcal{H})$ denote the set of boolean conjunctions of upper or lower bound expressions on clock variables in H; then $\rightarrow \subseteq L \times \mathcal{Z}(\mathcal{H}) \times A \times \wp(\mathcal{H}) \times L$. We write

$$l \xrightarrow{\zeta, a, \lambda} l'$$

for $(l, \zeta, a, \lambda, l') \in \rightarrow$, meaning that action a is enabled when the automation is at location l and the *guard* $\zeta \in \mathcal{Z}(\mathcal{H})$ is satisfied by the values of the variables in $\mathcal{H}$; then the automaton may *transit* to location l', *resetting* the clocks in subset $\lambda \subseteq \mathcal{H}$.

- $I : L \to \mathcal{Z}(H)$, $I(l)$ is a predicate that must be satisfied by the clocks while the automaton is at location l.

7.2.1 KLSV Structures

Kaynar, Lynch *et al.* [77], on the other hand, describe an automaton in a different way as a structure $(X, Q, \Theta, E, H, D, T)$. X is a set of *internal variables*; $Q \subseteq val(X)$ a set of *states* ($val(X)$ is the set of valuations – assignments of values to variables in X; thus a state is identified with a valuation of variables). $\Theta \subseteq Q$ is a set of *initial states* and E, H are sets of possible *external* and *internal actions*; $E \cap H = \emptyset$. This structure they call a *timed automaton*; in their *timed I/O automaton* E is further partitioned into a disjoint pair of subsets I of *input actions* and O of *output actions*, although these will not concern us much at present.

Let $A \triangleq E \cup H$. $D \subseteq Q \times A \times Q$ is the transition relation and T a set of possible *trajectories* in Q. These are "curves" in Q, mappings from a left-closed interval of the time line ($[0, b]$ or $[0, b)$ or $[0, \infty)$, $b \in \mathbb{R}$) to Q. T has certain closure properties:

- For each $q \in Q$ the *point trajectory* $[0, 0] \to Q,\ 0 \mapsto q$ is in T;
- a trajectory τ' is a *prefix* of τ, written $\tau' \leq \tau$, iff τ' is the restriction of τ to a left-closed subinterval of its domain.

  If $\tau' \leq \tau \in T$ then $\tau \in T$;
- if τ is a trajectory and $t \in dom(\tau)$ then the *suffix* $\tau \trianglerighteq t$ (to use the notation of [77]) is the mapping $u \mapsto \tau(t + u)$ defined for $u \geq 0$ such that $u + t \in dom(\tau)$, which defines a trajectory.

  Any suffix of a trajectory in T is in T;
- If τ is a *closed* trajectory – $dom(\tau) = [0, b]$, a finite closed interval, and τ' another trajectory, the *concatenation* $\tau \frown \tau'$ is the mapping $u \mapsto \tau(u)$ if $u \in [0, b]$ and $u \mapsto \tau'(u - b)$ if $u > b$ and $u - b \in dom(\tau')$[1].

[1] [77] proves a number of expected formal properties, such as $\tau \leq \upsilon$ iff $(\exists \tau')\tau \frown \tau' = \upsilon$, and that $\leq$ is a complete partial ordering.

If $\tau_k,\ k = 0, 1, 2, ...$ is a sequence of *closed* trajectories with domains $[0, b_k]$ and $(\forall k)\tau_k(b_k) = \tau_{k+1}(0)$, another sequence is defined recursively by $\tau'_0 \triangleq \tau_0$, $\tau'_{k+1} \triangleq \tau'_k \frown \tau_{k+1}$. This sequence $\{\tau_k\}$ is monotonic increasing with respect to the prefix relation (which is a complete partial ordering) $\leq$. The infinite concatenation $\tau_0 \frown \tau_1 \frown \tau_2 \frown ...$ is defined to be the $\leq$-least upper bound of the the directed set $\{\tau'_k | k = 0, 1, 2, ...\}$.

T is closed under the concatenation operation: Given a finite or infinite sequence of trajectories as above, in T, their concatenation is in T.

An *execution fragment* is a *hybrid sequence* $\tau_0 a_1 \tau_1 a_2 \tau_2 ...$ of alternating trajectories and actions ($a_i \in A$); a run or *execution* is an execution fragment such that $\tau_0(0) \in \Theta$. The sequence is infinite, or else the last term is a trajectory; and each trajectory other than the last is *closed*. There is a prefix ordering among such sequences: $\alpha \leq \beta$ iff $\alpha = \beta$ or α is finite, ending in a closed trajectory and this forms an initial segment of sequence β. Similarly, if α is a finite sequence $\tau_0 a_1 \tau_1 a_2 ... \tau_n$ with a closed final trajectory, the *concatenation* $\alpha \frown \beta$ is formed by concatenating τ_n with the initial trajectory of β. Kaynar, Lynch *et al.* [77] show that $\alpha \leq \beta$ iff $(\exists \beta')\alpha \frown \beta' = \beta$, and that $\leq$ among hybrid sequences renders them a complete partially ordered set, so that concatenation of an infinite hybrid sequence can be defined in a fashion similar to that of a sequence of trajectories.

7.2.2 Timed Automata and KLSV Structures

Our timed automaton $(L, l^0, A, \mathcal{H}, \rightarrow, I)$ can be represented as a structure of this type:

- Let $X = \mathcal{H} \cup \{\texttt{loc}\}$ where `loc` is a discrete variable whose values are the control locations L;
- $Q = val(X)$;
- Actions: $E = A, H = \emptyset$;
- Θ is the set of valuations which assign 0 to all clock variables ($\mathcal{H}$) and the value l^0 to `loc`.

- D is the set of $(x, a, x'), a \in A, x, x' \in val(X)$, such that $l \xrightarrow{\zeta,a,\lambda} l'$ in $(L, l^0, A, \mathcal{H}, \rightarrow, I)$, and $x(\texttt{loc}) = l, x \models \zeta, x'(\texttt{loc}) = l'$, $x' \models I(l')$, and $\forall h \in \lambda . x'(h) = 0$.

- The trajectories are constrained by the requirements that the discrete values of `loc` do not vary in time, the values of the clock variables have unit rate of change, and the location invariants $I(l)$ be satisfied.

See, for example, [77] pp. 38-40 (where our timed automata are called "Alur-Dill automata"). The KLSV structure so defined by Kaynar, Lynch *et al.* is functionally equivalent to the timed transition system of the timed automaton.

Alternatives to setting $E = A,\ H = \emptyset$ are $E = \emptyset,\ H = A$, or possibly partitioning A into $E \cup H$ so that E contains the edge labels intended to participate in synchronisations, while H contains those intended to be purely "local", $H = A - E$.

7.2.3 Composition

KLSV structures of the type described in [77] can be composed in parallel; we have already seen such a construction for simple timed automata. Briefly, $\mathbb{A}_i = (X_i, Q_i, \Theta_i, E_i, H_i, D_i, T_i)$, for $i = 1, 2$ are *compatible* iff $H_1 \cap (H_2 \cup E_2) = \emptyset$ and *vice versa* and $X_1 \cap X_2 = \emptyset$. In that case, the composite is $\mathbb{A}_1 \parallel \mathbb{A}_2 = (X, Q, \Theta, E, H, D, T)$ where

- $X = X_1 \cup X_2$,

- $Q = \{x \in val(X) : x \restriction X_i \in Q_i, i = 1, 2\}$

- $\Theta = \{x \in val(X) : x \restriction X_i \in \Theta_i, i = 1, 2\}$

- $E = E_1 \cup E_2,\ H = H_1 \cup H_2$,

- For $x, x' \in Q, a \in A \subseteq E \cup H,\ (x, a, x') \in D$ iff for $i = 1, 2$,
  either $a \in A_i = E_i \cup H_i$ and $(x \restriction X_i, a, x' \restriction X_i) \in D_i$;
  or $a \notin A_i$ and $x \restriction X_i = x' \restriction X_i$

- T consists of X-trajectories whose restrictions in X_i lie in T_i

This composition turns out to be equivalent to ours for structures derived from our automata. Consider two of our automata: $\mathbb{A}_i = (L_i, l_i^0, A_i, \mathcal{H}_i, \rightarrow_i, I_i), i = 1, 2$. Translating these into the formalism of [77] as above yields, for $i = 1, 2$,

- $X_i = \mathcal{H}_i \cup \{\mathtt{loc}_i\}$ where $\mathtt{loc}_i$'s values are the L_i;
- $Q_i = val(X_i)$;
- $E_i = A_i,\ H_i = \emptyset$;
- $x \in \Theta_i \ \Leftrightarrow\ \forall h \in \mathcal{H}_i.x(h) = 0\ \&\ x(\mathtt{loc}_i) = l_i^0$.
- $(x, a, x') \in D_i$ iff for some $l_i \xrightarrow{\zeta_i, a, \lambda_i}_i l_i'$ in $\mathbb{A}_i$, $x(\mathtt{loc}_i) = l_i$, $x'(\mathtt{loc}_i) = l_i',\ x \models \zeta_i$, and $x(h) = 0$ for $h \in \lambda_i$.

Forming the composite of [77],

- $X = \mathcal{H}_1 \cup \mathcal{H}_2 \cup \{\mathtt{loc}_1, \mathtt{loc}_2\}$,
- $Q = val(X)$
- $x \in \Theta$ iff for every $h \in \mathcal{H}_1 \cup \mathcal{H}_2, x(h) = 0$ and also $x(\mathtt{loc}_i) = l_i^0$ for $i = 1, 2$.
- $E = A_1 \cup A_2,\ H = \emptyset$,
- For $x, x' \in Q, a \in A_1 \cup A_2,\ (x, a, x') \in D$ iff for $i = 1, 2$, either $a \in A_i$ and $(x \upharpoonright X_i, a, x' \upharpoonright X_i) \in D_i$; or $a \notin A_i$ and $x \upharpoonright X_i = x' \upharpoonright X_i$.

The last item above is equivalent to: for $i = 1, 2$,

$$\begin{aligned} &\text{either } a \in A_i \text{ and for some } l_i \xrightarrow{\zeta_i, a, \lambda_i}_i l_i' \text{ in } \mathbb{A}_i,\ x(\mathtt{loc}_i) = l_i, \\ &\qquad x'(\mathtt{loc}_i) = l_i',\ x \models \zeta_i, \text{ and } x(h) = 0 \text{ for } h \in \lambda_i; \\ &\text{or } a \notin A_i \text{ and } x, x' \text{ agree on } \mathcal{H}_i \text{ and } x(\mathtt{loc}_i) = x'(\mathtt{loc}_i). \end{aligned} \tag{7.1}$$

By comparison, our parallel composite is

$$\mathbb{A}_1 \parallel \mathbb{A}_2 = (L_1 \times L_2, (l_1^0, l_2^0), A_1 \cup A_2, \rightarrow, I)$$

where
$I(l_1, l_2) = I_1(l_1) \wedge I_2(l_2)$ and
$(l_1, l_2) \xrightarrow{\zeta,a,\lambda} (l_1', l_2')$ iff one of the following obtains for some $l_1 \xrightarrow{\zeta_1,a,\lambda_1}_1 l_1'$ in $\mathbb{A}_1$ and/or some $l_2 \xrightarrow{\zeta_2,a,\lambda_2}_2 l_2'$ in $\mathbb{A}_2$:

$$a \in A_1 \cap A_2, \zeta = \zeta_1 \wedge \zeta_2, \lambda = \lambda_1 \cup \lambda_2 \tag{7.2}$$

$$a \in A_1 - A_2, \zeta = \zeta_1, \lambda = \lambda_1, l_2 = l_2' \tag{7.3}$$

$$a \in A_2 - A_1, \zeta = \zeta_2, \lambda = \lambda_2, l_1 = l_1' \tag{7.4}$$

If we identify $\{\texttt{loc}_1, \texttt{loc}_2\}$ with $\{\texttt{loc}\}$ where $x(\texttt{loc}) = (x(\texttt{loc}_1), x(\texttt{loc}_2))$ then we see that the two composite constructions are equivalent.

The definition of $(l_1, l_2) \xrightarrow{\zeta,a,\lambda} (l_1', l_2')$ is essentially equivalent to condition (7.1) above, bearing in mind that if $a \notin A_1, A_2$, the condition yields a trivial "identity transition".

Condition (7.2) gives "synchronising transitions" and conditions (7.3, 7.4) give "interleaving transitions".

7.2.4 UPPAAL Automata as KLSV Structures

An UPPAAL process template can be rendered as a KLSV-style timed automaton in a way that naturally generalises the definition of §7.2.2:

- Let $X = \mathcal{H} \cup \{\texttt{loc}\}$ where `loc` is, as before, a discrete variable whose values are the control locations of the template; but now $\mathcal{H}$ is the set of all variables of the process template.
- $Q = val(X)$ is now the set of all valuations of variables in X;
- Actions: E is the set of actions which synchronise: `c?` or `c!` where `c` is some channel identifier. Internal actions are anonymous in UPPAAL ("τ"): H is thus a singleton.

- Θ is the set of valuations which assign 0 to all variables in $\mathcal{H}$, or initialise them as prescribed by their UPPAAL declarations, and assign the value l^0 to `loc`.

- D is the set of $(x, a, x'), a \in E \cup H, x, x' \in val(X)$, such that there is an edge from location $x(\texttt{loc})$ to $x'(\texttt{loc})$ labelled a, and x satisfies the guard (if any) on the edge, and x' satisfies the invariant at $x'(\texttt{loc})$, and x' respects any updates that occur on the edge.

- The trajectories are constrained by the requirements that the discrete values of `loc` do not vary in time, neither do the values of non-clock variables, the values of the clock variables have unit rate of change, and the location invariants be always satisfied.

7.3 An Assume-Guarantee Theorem

7.3.1 Implementation Relationships

Kaynar, Lynch, Segala and Vaandrager in [77], [78] are interested in *implementation relationships* between their automata. Of particular interest is a relation $\mathbb{A}_1 \preceq \mathbb{A}_2$ between two automata of the $(X, Q, \Theta, E, H, D, T)$-type, KLSV-structures: every trace of $\mathbb{A}_1$ is also a trace of $\mathbb{A}_2$.

As seen above, in the parlance of Kaynar, Lynch *et al.*, an *execution fragment* of a (KLSV) automaton is a sequence $\tau_0, a_1, \tau_1, a_2, ...$of alternate *trajectories* and discrete actions. Each trajectory τ_k is a mapping of an interval of time into Q, and so tracks changes in values of the continuous variables as an interval of time passes. Each a_k is an action drawn from $E \cup H$. A *trace* is an execution restricted to the empty set of variables: that is, a sequence showing pure passages of time alternating with discrete actions.

$\mathbb{A}_1 \preceq \mathbb{A}_2$ therefore means, in effect, every possible behaviour (timed sequence of actions) of $\mathbb{A}_1$ is a possible behaviour of $\mathbb{A}_2$; or more succinctly, the behaviour *specified* by $\mathbb{A}_2$ is *implemented* by $\mathbb{A}_1$.

Kaynar, Lynch *et al.* call two of their (KLSV) automata *comparable* if they have the same external action labels (set E). Their I/O automata have E partitioned into input actions I and output actions O: in this case these two sets of action labels are the same if the automata are comparable. They call two automata *compatible* if the two sets of states X are disjoint, each set of internal actions H is disjoint from the other's $E \cup H$, and the output action labels O are disjoint.

7.3.2 Simulation and Trace Inclusion

Given comparable KLSV structures $\mathbb{A}_i = (X_i, Q_i, \Theta_i, E_i, H_i, D_i, T_i)$, $i = 1, 2$, a *simulation* is defined as a relation between states, $R \subseteq Q_1 \times Q_2$, analogously to §2.4.2:

- $\forall q_1 \in \Theta_1 \exists q_2 \in \Theta_2 : (q_1, q_2) \in R$.

- If $(q_1, q_2) \in R$ and $(q_1, a, q_1') \in D_1$ then there is an execution fragment in $\mathbb{A}_2$ consisting of the action a preceded by a closed trajectory *from*[2] q_1' and succeeded by a closed trajectory *to*[3] q_2' (say), and the trace in $\mathbb{A}_2$, the restriction of the trajectories to clocks, is the same as the trace in $\mathbb{A}_1$: action a preceded and succeeded by point trajectories; and $(q_1', q_2') \in R$.

- If $(q_1, q_2) \in R$ and τ is a closed trajectory in $\mathbb{A}_1$ from q_1, to q_1', say, then there is in $\mathbb{A}_2$ a closed trajectory with the same trace as τ, from q_2 to a state q_2' such that $(q_1', q_2') \in R$.

Kaynar, Lynch *et al.* term this a *forward simulation* – see [77] §4.5 (p41). They prove (Theorem 4.23, Corollary 4.24) that if such a forward simulation exists, then the traces of $\mathbb{A}_1$ are included in the traces of $\mathbb{A}_2$, i.e., $\mathbb{A}_1 \preceq \mathbb{A}_2$.

[2] By a trajectory τ *from* q we mean $\tau(0) = q$.

[3] Recall a closed trajectory τ is defined on a closed interval $[0, b]$; by a trajectory *to* q' we mean $\tau(b) = q'$

Proposition If we refer back to the representation of a simple timed automaton as a KLSV structure, subsection 7.2.2 we can infer a simulation relation exists between (the timed transition systems of) two timed automata, in the sense of §2.4.2, then again the traces of the first timed automaton are included in the traces of the second.

7.3.3 An Assume-Guarantee Theorem

Kaynar, Lynch *et al.* define a *Timed I/O Automaton* ([77], section 7) to be a structure consisting of what we have hitherto called a KLSV structure, $(X, Q, \Theta, E, H, D, T)$ together with a partition $E = I \cup O,\ I \cap O = \emptyset$ of the external actions into a set of *input actions* I and a set of output actions O. The structure must satisfy two additional axioms:

- An input-action enabling axiom: $\forall x \in Q\ \forall a \in I\ \exists x' : (x, a, x') \in D$;
- A time-passage-enabling axiom: For every $x \in Q$ there is a $\tau \in T$ from x such that *either* τ is infinite *or* τ is closed and is *to* a state where some internal (H) or output (O) action is enabled.

These structures are an interesting possible subject of future work, but what interests us at present is a form of assume-guarantee theorem which these authors state as Theorem 8.7:

Given automata $\mathbb{A}_1, \mathbb{A}_2, \mathbb{B}_1, \mathbb{B}_2$ such that

1. $\mathbb{A}_1, \mathbb{A}_2$ are comparable;
2. $\mathbb{B}_1, \mathbb{B}_2$ are comparable;
3. $\mathbb{A}_i$ is compatible with $\mathbb{B}_i$ $(i = 1, 2)$;
4. The set of traces of $\mathbb{A}_2$ and the set of traces of $\mathbb{B}_2$ are *limit-closed* (i.e. the set contains all sequences of which all finite prefixes are in the set);
5. Suffixing a trace over $\mathbb{A}_2$ with a trajectory over $\emptyset$ yields again a trace over $\mathbb{A}_2$; and similarly for $\mathbb{B}_2$;

IF $\mathbb{A}_1 \parallel \mathbb{B}_2 \preceq \mathbb{A}_2 \parallel \mathbb{B}_2$ *AND* $\mathbb{A}_2 \parallel \mathbb{B}_1 \preceq \mathbb{A}_2 \parallel \mathbb{B}_2$ *THEN* $\mathbb{A}_1 \parallel \mathbb{B}_1 \preceq \mathbb{A}_2 \parallel \mathbb{B}_2$.

7.3.4 UPPAAL Templates as TIOA

An UPPAAL process template can be construed as a timed I/O automaton in the sense of Kaynar, Lynch, *et al.* [77] provided it satisfies certain extra axioms. Let us define an UPPAAL process template to be an *UPPAAL I/O automaton* iff

1. (Input enabling axiom) For every synchronisation of the form `c?` and at every location other than a committed or urgent location there is an enabled edge from the location labelled `c?`.

2. (Time passage enabling axiom) At every location, *either* the invariant is satisfied for all time, *or* time passage proceeds to a state at which some output (`c!` for some `c`) or local (internal) action is enabled.

The assume-guarantee theorem of [77] can be stated and proved in terms of these structures. One obtains s KLSV-style I/O automaton by defining the KLSV structure as in §7.2.4 and additionally defining the set O of output actions as those actions of the structure which are `c!`-synchronisations for channels `c`, and the set I of input actions as the `c?`-synchronisations. The two conditions set out above are required in order to satisfy the input action enabling and time passage enabling axioms for KLSV-type I/O automata.

In order to state the theorem, first note that a *trace* as defined by [77] to be an execution restricted to the empty set of variables and external actions (cf §3.4.3) is, in the case of an UPPAAL template, simply an alternating sequence of time passages and discrete actions, starting with a time passage:

$$l_0 \xrightarrow{t_0} l_0 \xrightarrow{a_1} l_1 \xrightarrow{t_1} l_1 \xrightarrow{a_2} l_2 \xrightarrow{t_2} ...$$

The invariant at location l_n is satisfied throughout t_n. Location l_{n+1} carries a (timed) state which incorporates any update associated with action a_n. The timed state associated with l_n after time passage t_n satisfies any guard on a_n. Of course, the trace is the sequence $t_0 a_0 t_1 a_1 t_2 ...$; the locations are not part of the trace. A *closed* trace finishes with a time passage, or the sequence may go to infinity. A time passage of 0 corresponds to a *point trajectory* of c.

Also following these authors, let us call the set of traces of an UPPAAL I/O automaton *limit-closed* iff any sequence, all of whose finite prefixes are a trace, is itself a trace. Also say that two UPPAAL I/O automata are *comparable* if they have the same external actions (i.e. synchronisations) and *compatible* if the internal actions of either are disjoint from the actions of the other. The latter is what we require in order to be able to compose two automata. Let us also follow them in writing $\mathbb{A}_1 \preceq \mathbb{A}_2$ (when $\mathbb{A}_1, \mathbb{A}_2$ are comparable) to denote that traces of $\mathbb{A}_1 \subseteq$ traces of $\mathbb{A}_2$.

Composition of automata is *our* composition. In §7.2.3 it was seen to be equivalent to that of Kaynar, Lynch *et al.* and this is the case for UPPAAL automata too.

In the sequel, we speak of "UPPAAL I/O automata" while considering that an UPPAAL I/O automaton may in fact itself be a composite; so that the locations l_n referred to above are actually vectors.

7.3.4.1 Theorem

Let $\mathbb{A}_1$, $\mathbb{A}_2$ be comparable UPPAAL I/O automata, $\mathbb{B}_1$, $\mathbb{B}_2$ be comparable UPPAAL I/O automata, and suppose each $\mathbb{A}_i$ is compatible with each $\mathbb{B}_j$ $(i, j = 1, 2)$. Suppose further that

1. The set of traces of $\mathbb{A}_2$ is limit-closed, and the set of traces of $\mathbb{B}_2$ is limit-closed;

2. Concatenating a closed trace of $\mathbb{A}_2$ with further time passage yields again a trace of $\mathbb{A}_2$ (i.e., the invariants are satisfied) and likewise for $\mathbb{B}_2$;

THEN $\mathbb{A}_1 \parallel \mathbb{B}_2 \preceq \mathbb{A}_2 \parallel \mathbb{B}_2$ and $\mathbb{A}_2 \parallel \mathbb{B}_1 \preceq \mathbb{A}_2 \parallel \mathbb{B}_2$ imply $\mathbb{A}_1 \parallel \mathbb{B}_1 \preceq \mathbb{A}_2 \parallel \mathbb{B}_2$.

Remark This is exactly theorem 8.7 Kaynar, Lynch *et al.* [77] (see also §7.3.3), applied to UPPAAL I/O automata and stated in terms of these. It is instructive also to work through these authors' proof in terms of UPPAAL I/O automata.

Proof Suppose $\mathbb{A}_1 \parallel \mathbb{B}_2 \preceq \mathbb{A}_2 \parallel \mathbb{B}_2$ and $\mathbb{A}_2 \parallel \mathbb{B}_1 \preceq \mathbb{A}_2 \parallel \mathbb{B}_2$.

One first proves that any *closed* trace of $\mathbb{A}_1 \parallel \mathbb{B}_1$ is a trace also of $\mathbb{A}_2 \parallel \mathbb{B}_2$. This is proven by induction on the length of a trace

$$l_0 \stackrel{t_0}{\rightarrow} l_0 \stackrel{a_1}{\rightarrow} l_1 \stackrel{t_1}{\rightarrow} l_1 \stackrel{a_2}{\rightarrow} l_2 \stackrel{t_2}{\rightarrow} ...$$

The locations here are, of course vectors of $\mathbb{A}_1$ components and $\mathbb{B}_1$ components.

The base of the induction is a simple time transition $l_0 \stackrel{t_0}{\rightarrow} l_0$ in $\mathbb{A}_1 \parallel \mathbb{B}_1$. Then, t_0 satisfies the invariants the components of l_0 in $\mathbb{A}_1 \parallel \mathbb{B}_1$. From the comparability hypotheses we can infer that t_0 is also a trace in $\mathbb{A}_2 \parallel \mathbb{B}_2$.

Now suppose given a closed trace $\alpha_n = (l_0 \stackrel{t_0}{\rightarrow} ... \stackrel{t_{n-1}}{\rightarrow} l_{n-1} \stackrel{a_n}{\rightarrow} l_n)$ in $\mathbb{A}_1 \parallel \mathbb{B}_1$. Assume by induction hypothesis that the prefix $\alpha_{n-1} = (l_0 \stackrel{t_0}{\rightarrow} ... \stackrel{t_{n-1}}{\rightarrow} l_{n-1})$ in $\mathbb{A}_1 \parallel \mathbb{B}_1$ is actually a closed trace in $\mathbb{A}_2 \parallel \mathbb{B}_2$. To establish the induction, that α_n is a trace in $\mathbb{A}_2 \parallel \mathbb{B}_2$ there are the following cases to consider.

1. a_n is an output action `c!` of (a component of) $\mathbb{A}_1$ and $t_n = 0$. In this case, α_n projects on $\mathbb{A}_1$ to a trace $\alpha_n \upharpoonright \mathbb{A}_1$ and also α_{n-1} by induction hypothesis, projects on $\mathbb{B}_2$ to a trace of $\mathbb{B}_2$. Because $\mathbb{A}_1, \mathbb{B}_1$ are compatible and $\mathbb{B}_1, \mathbb{B}_2$ are comparable, either a_n is an *input* action (`c?`) of $\mathbb{B}_2$ or it is not an action of $\mathbb{B}_2$ at all. In the former case, the input enabling axiom implies that $\mathbb{B}_2$ has an action a_n synchronising with the $\mathbb{A}_1$ action a_n (i.e. `c!`, `c?`) whence projecting α_n to the external actions $\mathbb{B}_2$ gives a trace of $\mathbb{B}_2$. In the latter case, α_n and α_{n-1} agree on $\mathbb{B}_2$. Either way, pasting these traces on $\mathbb{A}_1$, $\mathbb{B}_2$ together, we have a trace in $\mathbb{A}_1 \parallel \mathbb{B}_2$. By the hypothesis of the theorem, $\mathbb{A}_1 \parallel \mathbb{B}_2 \preceq \mathbb{A}_2 \parallel \mathbb{B}_2$, this is a trace in $\mathbb{A}_2 \parallel \mathbb{B}_2$.

2. a_n is an output action `c!` of (a component of) $\mathbb{B}_1$ and $t_n = 0$. This is symmetrical with the previous case, using hypothesis $\mathbb{A}_2 \parallel \mathbb{B}_1 \preceq \mathbb{A}_2 \parallel \mathbb{B}_2$ rather than $\mathbb{A}_1 \parallel \mathbb{B}_2 \preceq \mathbb{A}_2 \parallel \mathbb{B}_2$.

3. a_n is an input action `c?` of both $\mathbb{A}_1$ and $\mathbb{B}_1$, and $t_n = 0$. By induction hypothesis, α_{n-1} is a trace of $\mathbb{A}_2 \parallel \mathbb{B}_2$. Projecting onto the two component subsystems, we have projections of α_{n-1} which are traces of $\mathbb{A}_2$

and of $\mathbb{B}_2$. Let α be a run of $\mathbb{A}_2$ whose trace is the projection of α_{n-1} on $\mathbb{A}_2$. Because $\mathbb{A}_1$ and $\mathbb{A}_2$ are comparable, a_n is an input action of $\mathbb{A}_2$. By the input enabling axiom there is an input action labelled a_n from the projection of l_{n-1} on $\mathbb{A}_2$. It follows that the projection of α_n on $\mathbb{A}_2$ is a trace of $\mathbb{A}_2$.

Repeating this reasoning swapping the roles of $\mathbb{A}_2$ and $\mathbb{B}_2$, we see that also the projection of α_n on $\mathbb{B}_2$ is a trace of $\mathbb{B}_2$. It follows the α_n is a trace of $\mathbb{A}_2 \parallel \mathbb{B}_2$.

4. a_n is an input action of $\mathbb{A}_1$ but not an action of $\mathbb{B}_1$, and $t_n = 0$. By induction hypothesis, α_{n-1} is a trace of $\mathbb{A}_2 \parallel \mathbb{B}_2$. As in the previous case, it projects to traces in $\mathbb{A}_2$ and in $\mathbb{B}_2$. Let α be a run of $\mathbb{A}_2$ whose trace is the projection of α_{n-1} on $\mathbb{A}_2$. Because $\mathbb{A}_1$ and $\mathbb{A}_2$ are comparable, a_n is an input action of $\mathbb{A}_2$. By the input enabling axiom we can concatenate to α an action $l_{n-1} \restriction \mathbb{A}_2 \xrightarrow{a_n} l_n \restriction \mathbb{A}_2$ (where $\restriction$ denotes projection). So α_n projects to a trace of $\mathbb{A}_2$.

   $\mathbb{B}_1$ and $\mathbb{B}_2$ are comparable and a_n is *not* an action of $\mathbb{B}_1$, hence it is also *not* an action of $\mathbb{B}_2$. So α_n, α_{n-1} agree on $\mathbb{B}_2$. Combining this information with the previous paragraph we obtain α_n as a trace in $\mathbb{A}_2 \parallel \mathbb{B}_2$.

5. a_n is an input action of $\mathbb{B}_1$ but not an action of $\mathbb{A}_1$, and $t_n = 0$. This is symmetric with the previous case, swapping $\mathbb{A}$s and $\mathbb{B}$s.

6. $t_n > 0$. We can assume $... \xrightarrow{a_n} l_n \xrightarrow{0} l_n$ in $\mathbb{A}_2 \parallel \mathbb{B}_2$ by induction hypothesis. Project this onto each of $\mathbb{A}_2$, $\mathbb{B}_2$ and apply premiss 2 of the theorem to each projected trace. This yields $... \xrightarrow{a_n} l_n \xrightarrow{t_n} l_n$ projecting to a trace on each of $\mathbb{A}_2$, $\mathbb{B}_2$. It follows that in its totality it is a trace of $\mathbb{A}_2 \parallel \mathbb{B}_2$.

This completes the proof that any *closed* trace of $\mathbb{A}_1 \parallel \mathbb{B}_1$ is a (closed) traces of $\mathbb{A}_2 \parallel \mathbb{B}_2$.

We infer the same inclusion for *all* traces (including infinite ones) by lifting closed prefixes of a trace in $\mathbb{A}_1 \parallel \mathbb{B}_1$ to $\mathbb{A}_2 \parallel \mathbb{B}_2$, projecting each of

these onto each of $\mathbb{A}_2$, $\mathbb{B}_2$ and on each component, invoking limit-closure (theorem premiss 1) to infer a sequence which projects on each of $\mathbb{A}_2$, $\mathbb{B}_2$ to a trace, and which is therefore a trace of $\mathbb{A}_2 \parallel \mathbb{B}_2$.

7.4 A Compositionality Theorem for *Our* Timed Automata

This section and the next explore the possibility of a similar assume-guarantee theorem for *our* timed automata, and also (next section) for UPPAAL automata. There are theorems asserting inclusion of *runs*, not merely traces; but we shall see rather strong assumptions are required regarding the nature of synchronised actions, which may limit the usefulness of these theorems.

7.4.1 Runs, Traces

A *trace* in the sense of Kaynar, Lynch *et al.* turns out in the case of our automata to be just a run of the type described in this subsection. When the action labels A of two of our automata, $\mathbb{A}_1, \mathbb{A}_2$ are the same, the relation $\mathbb{A}_1 \preceq \mathbb{A}_2$ says just that every possible run of $\mathbb{A}_1$ is a possible run of $\mathbb{A}_2$.

A *run* of $\mathbb{A} = (L, l^0, A, \mathcal{H}, \rightarrow, I)$ is a sequence of alternate time passages $t_k \in \mathbb{R}$ and *discrete actions* $a_k \in A$:

$$(v^0, l^0) \xrightarrow{t_0} (v^0 + t_0, l^0) \xrightarrow{a_1} (v^1, l^1) \xrightarrow{t_1} (v^1 + t_1, l^1) \xrightarrow{a_2} ... \xrightarrow{a_k} (v^k, l^k) \xrightarrow{t_k} ... \quad (7.5)$$

Each *state* on the way consists of a control location $l^k \in L$ of the underlying automaton, and a *valuation*, $v^k \in val(\mathcal{H})$, an assignment of values to the variables in $\mathcal{H}$ (cf $val(X)$ in the formalism of Kaynar, Lynch *et al.*). v^0 is the 0 valuation, $v^0(x) = 0$, and $v^k + t$ is the valuation which assigns the value $v^k(x) + t$ to variable x.

The arrows labelled with a real numbers represent time passage; t_k represents time passage at location l^k. The invariant must remain true during time spent at a location: $\forall k. \forall t \in [0, t_k]. (v^k + t) \models I(l^k)$.

The arrows labelled with discrete actions arise from transitions of $\mathbb{A}$. The arrow $(v^{k-1}+t_{k-1}, l^{k-1}) \xrightarrow{a_k} (v^k, l^k)$ arises from $\mathbb{A}$-transition $l^{k-1} \xrightarrow{\zeta, a_k, \lambda} l^k$ where $v^{k-1}+t_{k-1} \models \zeta$ and $v^k = (v^{k-1}+t_{k-1})[\lambda]$, the valuation assigning 0 to variables in λ and otherwise agreeing with $v^{k-1}+t_{k-1}$. Note that the previous paragraph ensures that the invariant will be satisfied in the target state.

Now let

$$\mathbb{A} = (L, l^0, A, \mathcal{H}, \rightarrow_A, I) \tag{7.6}$$

$$\mathbb{B} = (M, m^0, B, \mathcal{K}, \rightarrow_B, J) \tag{7.7}$$

denote two automata. Assume $\mathcal{H} \cap \mathcal{K} = \emptyset$ – the clock variables are disjoint. Thus, a valuation in $\mathbb{A} \parallel \mathbb{B}$ is just the union of an $\mathbb{A}$-valuation and a $\mathbb{B}$-valuation. There is no overlap to worry about.

A run of $\mathbb{A} \parallel \mathbb{B}$ is a sequence of the form

$$\begin{aligned}(v^0, l^0, m^0) \xrightarrow{t_0} (v^0+t_0, l^0, m^0) \xrightarrow{a_1} (v^1, l^1, m^1) \xrightarrow{t_1} (v^1+t_1, l^1, m^1) \\ \xrightarrow{a_2} ... \xrightarrow{a_k} (v^k, l^k, m^k) \xrightarrow{t_k} (v^k+t_k, l^k, m^k) \rightarrow ...\end{aligned} \tag{7.8}$$

where

- $t_k \in \mathbb{R}, a_k \in A \cup B$ and
- $\forall k. \forall t \in [0, t_k]. (v^k + t) \models I(l^k) \wedge J(m^k)$; and
- For all k, one of the following obtains: *EITHER*
  - (a synchronised transition) $a_k \in A \cap B$ and there are transitions $l^{k-1} \xrightarrow{\zeta_A, a_k, \lambda_A} l^k$ in $\mathbb{A}$ and $m^{k-1} \xrightarrow{\zeta_B, a_k, \lambda_B} m^k$ in $\mathbb{B}$, and
    $(v^{k-1}+t_{k-1}) \models \zeta_A \wedge \zeta_B$ and
    $v^k = (v^{k-1}+t_{k-1})[\lambda_A \cup \lambda_B] = (v^{k-1}+t_{k-1})[\lambda_A][\lambda_B]$; *OR*
  - (an interleaving transition) $a_k \in A - B$ and
    there is a transition $l^{k-1} \xrightarrow{\zeta, a_k, \lambda} l^k$ in $\mathbb{A}$; and
    $m^{k-1} = m^k$, and
    $(v^{k-1}+t_{k-1}) \models \zeta$ and $v^k = (v^{k-1}+t_{k-1})[\lambda]$; *OR*

– (an interleaving transition) $a_k \in B - A$ and
$l^{k-1} = l^k$, and
there is a transition $m^{k-1} \stackrel{\zeta, a_k, \lambda}{\longrightarrow} m^k$ in $\mathbb{B}$; and
$(v^{k-1} + t_{k-1}) \models \zeta$ and $v^k = (v^{k-1} + t_{k-1})[\lambda]$

7.4.2 Theorem

Let $\mathbb{A}_1, \mathbb{A}_2$ be automata with a common action alphabet, $\mathbb{B}_1, \mathbb{B}_2$ be automata with a common action alphabet, and suppose $\mathbb{A}_i$ is compatible with $\mathbb{B}_j$ for $i, j = 1, 2$ – the clocks and locations of $\mathbb{A}_i$, $\mathbb{B}_j$ are disjoint. Suppose that

$$\mathbb{A}_1 \parallel \mathbb{B}_2 \preceq \mathbb{A}_2 \parallel \mathbb{B}_2 \tag{7.9}$$

$$\mathbb{A}_2 \parallel \mathbb{B}_1 \preceq \mathbb{A}_2 \parallel \mathbb{B}_2 \tag{7.10}$$

Then $\mathbb{A}_1 \parallel \mathbb{B}_1 \preceq \mathbb{A}_2 \parallel \mathbb{B}_2$.

To prove this, we need to check, under the stated premisses, that any run of $\mathbb{A}_1 \parallel \mathbb{B}_1$ is also a run in $\mathbb{A}_2 \parallel \mathbb{B}_2$.

The argument will be an induction on the length (number of actions) of a run. First, Consider a "trivial run" of $\mathbb{A}_1 \parallel \mathbb{B}_1$,

$$(0, l^0, m^0) \stackrel{t_0}{\rightarrow} (0 + t_0, l^0, m^0) \tag{7.11}$$

where $t_0 \in \mathbb{R}$, 0 is the zero valuation and l^0, m^0 are the initial locations of $\mathbb{A}_1, \mathbb{B}_1$.

Without loss of generality, in view of (7.9, 7.10) we can assume l^0, m^0 are also the respective initial locations of $\mathbb{A}_2, \mathbb{B}_2$ – re-label if necessary. Let $\alpha_1, \alpha_2, \beta_1, \beta_2$ denote the invariants at l^0 in $\mathbb{A}_1, \mathbb{A}_2$ and at m^0 in $\mathbb{B}_1, \mathbb{B}_2$. Then (7.11) implies that

$$\forall t < t_0 : (0 + t) \models \alpha_1 \wedge \beta_1$$

We aim to deduce that $(0 + t) \models \alpha_2 \wedge \beta_2$.

Now, the $\mathbb{A}_i$-variables are disjoint from the $\mathbb{B}_j$-variables. Let v_1 denote a valuation which agrees with $(0+t)$ on the $\mathbb{A}$-variables, and which is 0 on the $\mathbb{B}$-variables. We can assume with no loss of generality that $0 \models \beta_2$.

Then $v_1 \models \alpha_1 \wedge \beta_2$. It follows from (7.9) that $v_1 \models \alpha_2 \wedge \beta_2$. In particular, $v_1 \models \alpha_2$.

Therefore, by construction of v_1, $(0+t) \models \alpha_2$.

By a similar argument employing premiss (7.10), $(0+t) \models \beta_2$.

Applying this to all $t \in [0, t_0]$ we can deduce that the time delay (7.11) is actually a run in $\mathbb{A}_2 \parallel \mathbb{B}_2$.

Now, let

$$(0, l^0, m^0) \xrightarrow{t_0} (0+t_0, l^0, m^0) \xrightarrow{a_1} \ldots \xrightarrow{t_{n-1}} (v^{n-1}+t_{n-1}, l^{n-1}, m^{n-1}) \xrightarrow{a_n} (v^n, l^n, m^n) \xrightarrow{t_n} (v^n+t_n, l^n, m^n) \quad (7.12)$$

be a run of "length" n in $\mathbb{A}_1 \parallel \mathbb{B}_1$. Assume as induction hypothesis that all runs of length shorter than n in $\mathbb{A}_1 \parallel \mathbb{B}_1$ map into $\mathbb{A}_2 \parallel \mathbb{B}_2$. We shall abuse notation by using the same symbols l^k, m^k $(k < n)$ for "corresponding" locations in $\mathbb{A}_2 \parallel \mathbb{B}_2$.

Then the part of (7.12) up to $(v^{n-1}+t_{n-1}, l^{n-1}, m^{n-1})$ is already a run in $\mathbb{A}_2 \parallel \mathbb{B}_2$.

At each stage, $k < n$, $v^k = v^{k-1} + t_{k-1}$ with the resets of a_k applied.

Now we have to consider three possibilities for the transition:

$$(v^{n-1}+t_{n-1}, l^{n-1}, m^{n-1}) \xrightarrow{a_n} (v^n, l^n, m^n)$$

We could have (1): $a_n \in A_1 - B_1$, an "interleaving" transition of $\mathbb{A}_1$, or (2): $a_n \in B_1 - A_1$,an interleaving transition of $\mathbb{B}_1$, or (3): $a_n \in A_1 \cap B_1$, a synchronizing transition.

First, suppose a_n to be an "interleaving" transition of $\mathbb{A}_1$. Thus,

$$m^n = m^{n-1},$$
$$v^n = v^{n-1} + t_{n-1}$$

with resets on $\mathbb{A}$-variables only.

This lifts to $\mathbb{A}_1 \parallel \mathbb{B}_2$ since m^n "is" already in $\mathbb{B}_2$. (Abuse of notation again: more precisely, there is a corresponding location of $\mathbb{B}_2$ which we can label m^n.)

Consider the time delay $(v^n, l^n, m^n) \xrightarrow{t_n} (v^n + t_n, l^n, m^n)$ in $\mathbb{A}_1 \parallel \mathbb{B}_1$. Let α_1, β_1 be the invariants in $\mathbb{A}_1, \mathbb{B}_1$ at l^n, m^n. Then,

$$\forall t < t_n : (v^n + t) \models \alpha_1 \wedge \beta_1$$

Let α_2, β_2 be the invariants in $\mathbb{A}_2, \mathbb{B}_2$ at l^n, m^n (abuse of notation again) and for arbitrary $t < t_n$ define the valuation w_t on the $\mathbb{A}$-variables as $v^n + t$ and on the $\mathbb{B}$-variables as v^n. Then $w_t \models \alpha_1$ and also $w_t \models \beta_2$ because $m^n = m^{n-1}$ and we can appeal to the induction hypothesis.

Now,

$$\forall t < t_n : w_t \models \alpha_1 \wedge \beta_2$$

exhibits t_n as a possible time delay "run" in $\mathbb{A}_1 \parallel \mathbb{B}_2$; hence by premiss (7.9), as a time delay in $\mathbb{A}_2 \parallel \mathbb{B}_2$. It follows (the reasoning is the same as in the induction base case) that

$$\forall t < t_n : w_t \models \alpha_2 \wedge \beta_2$$

In particular, $\forall t < t_n : w_t \models \alpha_2$. So, since w_t agrees with $v^n + t$ on $\mathbb{A}$-variables, $\forall t < t_n : (v^n + t) \models \alpha_2$.

By a similar argument swapping the roles of $\mathbb{A}_i, \mathbb{B}_i$ and employing premiss (7.10), $\forall t < t_n : (v^n + t) \models \beta_2$.

Thus t_n is a permissible time delay in $\mathbb{A}_2 \parallel \mathbb{B}_2$ and the segment

$$... \xrightarrow{a_n} (...) \xrightarrow{t_n} (...)$$

extends the "tail" of (7.12) to a run in $\mathbb{A}_2 \parallel \mathbb{B}_2$.

A similar argument lifts (7.12) to a run in $\mathbb{A}_2 \parallel \mathbb{B}_2$ in the case of the second possibility: that a_n is an interleaving transition of $\mathbb{B}_1$.

Now consider the third possibility: $a_n \in A_1 \cap B_1$, a_n is a synchronizing transition of $\mathbb{A}_1 \parallel \mathbb{B}_1$. In particular, there are transitions

$$l^{n-1} \xrightarrow{\zeta_1, a_n, \lambda_1} l^n \text{ in } \mathbb{A}_1 \tag{7.13}$$

$$m^{n-1} \xrightarrow{\eta_1, a_n, \mu_1} m^n \text{ in } \mathbb{B}_1 \tag{7.14}$$

where $(v^{n-1} + t_{n-1}) \models \zeta_1 \wedge \eta_1$ (the guards of a_n in $\mathbb{A}_1$ and $\mathbb{B}_1$) and $v^n = (v^{n-1} + t_{n-1})[\lambda_1 \cup \mu_1] \models \alpha_1 \wedge \beta_1$, the invariants at l^n, m^n.

Also, in view of the time delay $(v^n, l^n, m^n) \xrightarrow{t_n} (v^n + t_n, l^n, m^n)$ in $\mathbb{A}_1 \parallel \mathbb{B}_1$, we have $\forall t < t_n : (v^n + t) \models \alpha_1 \wedge \beta_1$.

Now define a valuation w in $\mathbb{A}_1 \parallel \mathbb{B}_2$ to agree with v^n on $\mathbb{A}_1$ and $v^{n-1} + t_{n-1}$ on $\mathbb{B}_2$. Remember, by the induction hypothesis, the "tail" of (7.12) lifts from $\mathbb{A}_1 \parallel \mathbb{B}_1$ to $\mathbb{A}_2 \parallel \mathbb{B}_2$; in particular, $v^{n-1} + t_{n-1}$ makes sense in $\mathbb{B}_2$.

This w gives a transition enabled in $\mathbb{A}_1 \parallel \mathbb{B}_2$:

$$(v^{n-1} + t_{n-1}, l^{n-1}, m^{n-1}) \xrightarrow{a_n} (w, l^n, m^{n-1})$$

which by premiss (7.9) lifts to $\mathbb{A}_2 \parallel \mathbb{B}_2$

It follows from this that the guard of a_n in $\mathbb{A}_2$ is satisfied by $v^{n-1} + t_{n-1}$ and the invariant at l^n in $\mathbb{A}_2$ by w, hence (by definition of w) by v^n.

Now apply the same argument with $\mathbb{A}_i, \mathbb{B}_i$ reversed to obtain an edge in $\mathbb{B}_2$ corresponding to a_n, whose guard is satisfied by $v^{n-1} + t_{n-1}$ and such that

the invariant at its target (corresponding to l^{n-1}, m^n in $\mathbb{A}_2 \parallel \mathbb{B}_1$) is satisfied by v^n. For this we need to define w to be like v^n on $\mathbb{B}_1$ and $v^{n-1} + t_{n-1}$ on $\mathbb{A}_2$.

The result is edges in $\mathbb{A}_2, \mathbb{B}_2$ corresponding to (7.13, 7.14) whose guards and target invariants are satisfied, and so the synchronized a_n is lifted to $\mathbb{A}_2 \parallel \mathbb{B}_2$.

The time delay transition t_n is lifted to $\mathbb{A}_2 \parallel \mathbb{B}_2$ using exactly the same reasoning as before.

Thus the run (7.12) is in $\mathbb{A}_2 \parallel \mathbb{B}_2$ in the case of a synchronised transition and the induction is complete.

7.5 Compositionality Theorem for UPPAAL Automata

It is interesting to see if the theorem of 7.4.2 can be extended to UPPAAL systems, whose semantics are described in 5.2.2. Again, the result will apply to parallel compositions of automata (UPPAAL process instances) with no shared variables. For our purposes, therefore, we shall suppose an UPPAAL System with no global variables apart from the *channels* used to mediate synchronisation, and just two process instances. By *variables* in the sequel, we mean just local variables within process instances.

7.5.1 Runs of the UPPAAL System

The first point to note is that a run as described by (7.5) above is the same in UPPAAL *except* for the following points.

- The component $\mathcal{H}$ is, for each *type*, a set of variables of that type;
- Variables of all types may participate in transition guards and location invariants;

- The valuations v assign values to all the variables; for $t \in \mathbb{R}$, valuation $v + t$ advances the *clocks* but leaves other values unchanged;

- A transition is not generally decorated with an action label (although a synchronisation label could be pressed into service) but the update λ is now a set of UPPAAL *assignments* to local variables rather than just a set of clocks to assigned 0.

- A discrete action in the run is as before an interleaving transition or several component actions in synchrony; but now the UPPAAL semantic rules of 5.2.2 decide which kind of action occurs when. UPPAAL priorities have not been employed in this work and will therefore not be mentioned in the following.

Consider now a run of a system $\mathbb{A}_1 \parallel ... \parallel \mathbb{A}_n$ generalising (7.6)(7.7), with $\mathbb{A}_i,\ i = 1...n$ the component UPPAAL processes. Unlike in the case of KLSV *traces*, we are now considering inclusion of *runs* without necessarily restricting to external actions and to the empty set of variables. It is useful to consider several processes rather than just two when considering the semantics of broadcast synchronisations. Recall that internal actions in UPPAAL are anonymous ("τ"); the action labels in A, B may be "borrowed" to denote synchronisations $c!,\ c?$, etc. Again we have a sequence of the form

$$\begin{aligned}(v^0, (l_i^0)_{i=1...n}) \xrightarrow{t_0} (v^0 + t_0, (l_i^0)) \xrightarrow{a_1} (v^1, (l_i^1)) \xrightarrow{t_1} (v^1 + t_1, (l_i^1)) \\ \xrightarrow{a_2} ... \xrightarrow{a_k} (v^k, (l_i^k)) \xrightarrow{t_k} (v^k + t_k, (l_i^k)) \rightarrow ...\end{aligned} \tag{7.15}$$

where

- $(l_i^k)_{i=1...n}$, or by slight abuse of notation, (l_i^k), is an abbreviation for a vector of locations $(l_1^k, l_2^k, l_3^k, ..., l_n^k)$;

- The l_i^0 are the initial locations of the component processes; $l_i^k,\ i = 1...n$ are their subsequent locations;

- $t_k \in \mathbb{R}$;

- a_k is normally in UPPAAL anonymous (a "τ" action), but we may borrow it to use as a synchronisation label;
- $\forall k.\forall t \in [0, t_k].(v^k + t) \models \bigwedge_{i=1}^{m} I_i(l_i^k)$, the invariants at (l_i^k); and
- For all k, one of the following obtains:

  1. There is an internal transition arising from a transition $l_i^{k-1} \xrightarrow{\zeta, a_k, \lambda} l_i^k$ on one component $\mathbb{A}_i$: a_k is "τ" (denoted "*" by nothing at all in the UPPAAL Help),
  $(\forall j \neq i) l_j^{k-1} = l_j^k$,
  $(v^{k-1} + t_{k-1}) \models \zeta$,
  $v^k = (v^{k-1} + t_{k-1})[\lambda]$, the result of applying the UPPAAL assignment(s) λ to the valuation $v^{k-1} + t_{k-1}$,
  $v^k \models I_i(l_i^k)$,
  either l_i^{k-1} is committed (in the UPPAAL sense) or $(\forall j \neq i) l_j^{k-1}$ is *not* committed.
  2. There is a binary synchronization: a_k denotes a binary *channel* and there are transitions on *different* processes i, j, $l_i^{k-1} \xrightarrow{\zeta_i, a_k!, \lambda_i} l_i^k$ in $\mathbb{A}_i$ and $l_j^{k-1} \xrightarrow{\zeta_j, a_k?, \lambda_j} l_j^k$ in $\mathbb{A}_j$,
  $(\forall h \neq i, j) l_h^{k-1} = l_h^k$,
  $(v^{k-1} + t_{k-1}) \models \zeta_i \wedge \zeta_j$,
  $v^k = (v^{k-1} + t_{k-1})[\lambda_i][\lambda_j]$ (Note that the UPPAAL assignment(s) of the !-edge are applied *before* the assignment(s) of the ?-edge),
  $v^k \models I_i(l_i^k) \wedge I_j(l_j^k)$,
  either l_i^{k-1}, l_j^{k-1} are both committed or $(\forall h \neq i, j) l_h^{k-1}$ is *not* committed.
  3. There is a broadcast synchronisation: a_k denotes a broadcast *channel* and there are transitions on *different* processes $i, j_1, ..., j_m$, $l_i^{k-1} \xrightarrow{\zeta_i, a_k!, \lambda_i} l_i^k$ in $\mathbb{A}_i$ and $l_{j_h}^{k-1} \xrightarrow{\zeta_{j_h}, a_k?, \lambda_{j_h}} l_{j_h}^k$ in $\mathbb{A}_{j_h}$ for $h = 1...m$, with $j_1, ..., j_m$ indexed in such a way that the $\mathbb{A}_{j_h}$ are numbered in the order they occur in the UPPAAL *system definition*,
  $(\forall h \neq i, j_1, ..., j_m) l_h^{k-1} = l_h^k$,
  $(v^{k-1} + t_{k-1}) \models \zeta_i \wedge \zeta_{j_1} \wedge ... \wedge \zeta_{j_m}$,

Every edge from a location other than $l_i, l_{j_1}, ..., l_{j_m}$ either has no synchronisation label $a_k?$, or the valuation $(v^{k-1} + t_{k-1})$ does *not* satisfy its guard, $v^k = (v^{k-1} + t_{k-1})[\lambda_i][\lambda_{j_1}]...[\lambda_{j_m}]$ (note the order in which the assignments are applied),
$v^k \models I_i(l_i^k) \wedge I_{j_1}(l_{j_1}^k) \wedge ... \wedge I_{j_m}(l_{j_m}^k)$,
either at least one of $l_i^{k-1},\ l_{j_1}^{k-1}, ..., l_{j_m}^{k-1}$ is committed or *no* location other than these is committed.

The assume-guarantee theorem 7.4.2 relied on a rather liberal synchronisation semantics; UPPAAL takes a rather stricter view of enforcing synchronisation. In view of this, the UPPAAL version of the assume-guarantee theorem requires stronger hypotheses.

7.5.2 Theorem

Let $\mathbb{A}_1, ..., \mathbb{A}_n$ be UPPAAL-type timed automata comprising a composite UPPAAL system (not employing priorities or global variables other than channels) and suppose each component $\mathbb{A}_i$ can be replaced by a component $\mathbb{A}'_i$ with the same synchronisations. Suppose further that if $i \neq j$, the local variables of $\mathbb{A}_i,\ \mathbb{A}'_i$ are disjoint from those of $\mathbb{A}_j,\ \mathbb{A}'_j$. In the sequel, let $\prod$ denote a parallel product[4].

If (a) for $i = 1, ..., n$,

$$\left(\prod_{1\leq j<i} \mathbb{A}'_j\right) \parallel \mathbb{A}_i \parallel \left(\prod_{i<j\leq n} \mathbb{A}'_j\right) \preceq \prod_{i=1}^{n} \mathbb{A}'_i, \tag{7.16}$$

and (b) whenever a subset of indices $I = \{i_1...i_r\}$ identifies system components which participate in a synchronised action,

$$\prod_{i=1}^{n} \mathbb{A}''_i \preceq \prod_{i=1}^{n} \mathbb{A}'_i \tag{7.17}$$

where $\mathbb{A}''_i$ is $\mathbb{A}_i$ when $i \in I$ and $\mathbb{A}'_i$ otherwise;

[4]e.g. $\prod_{i=1}^{n} \mathbb{A}_i$ for $\mathbb{A}_1 \parallel ... \parallel \mathbb{A}_n$

then

$$\prod_{i=1}^{n} \mathbb{A}_i \preceq \prod_{i=1}^{n} \mathbb{A}'_i \tag{7.18}$$

Proof. To show any trace of the product on the left of (7.18) is also a trace of the product on the right, we reason by induction on the length (the number of actions) of a trace.

First, consider a "trivial run" of $\mathbb{A}_1 \parallel ... \parallel \mathbb{A}_n$,

$$(v^0, (l_i^0)_{i=0...n}) \xrightarrow{t_0} (v^0 + t_0, (l_i^0)_{i=0...n}) \tag{7.19}$$

where $t_0 \in \mathbb{R}$, v^0 is the valuation which zeros all clocks and initialises local variables, and $(l_i^0)_{i=0...n}$ is the vector of initial locations of $\prod_{i=1}^{n} \mathbb{A}_i$. In the sequel such a vector of locations $(l_i)_{i=1...n}$ will be abbreviated $\overrightarrow{l}$.

Without loss of generality, in view of (7.16), we can assume the $\overrightarrow{l}^0$ are also the respective initial locations of $\prod_{i=0}^{n} \mathbb{A}'_i$ – re-label if necessary. Let α_i, α'_i denote the invariants at l_i^0 in $\mathbb{A}_i, \mathbb{A}'_i$. Then (7.19) implies that

$$\forall t < t_0 : (v^0 + t) \models \bigwedge_i \alpha_i$$

We aim to deduce that $(v^0 + t) \models \bigwedge_i \alpha'_i$.

Now, the variables of $\mathbb{A}_i, \mathbb{A}'_i$ are disjoint from those of $\mathbb{A}_j, \mathbb{A}'_j$ when $i \neq j$. For $i = 1, ..., n$, let v_i denote a valuation which agrees with $(v^0 + t)$ on the $\mathbb{A}_i, \mathbb{A}'_i$-variables, and which is as v^0 on the $\mathbb{A}_j, \mathbb{A}'_j$-variables for all $j \neq i$. We can assume with no loss of generality that $v^0 \models \alpha'_j$.

Then $v_i \models \alpha_i \wedge \bigwedge_{j \neq i} \alpha'_j$. It follows from (7.16) that $v_i \models \bigwedge_j \alpha'_j$. In particular, $v_i \models \alpha'_i$.

Therefore, by construction of v_i, $(v^0 + t) \models \alpha'_i$.

$$\forall t < t_0 : (v^0 + t) \models \bigwedge_i \alpha'_i$$

Applying this for all i and to all $t \in [0, t_0]$ we can deduce that the time

delay (7.19) is actually a run in $\prod_{i=1}^{n} \mathbb{A}'_i$.

Now, let

$$(v^0, \overrightarrow{l}^0) \xrightarrow{t_0} (v^0 + t_0, \overrightarrow{l}^0) \xrightarrow{a_1} \ldots \xrightarrow{t_{m-1}} (v^{m-1} + t_{m-1}, \overrightarrow{l}^{m-1}) \xrightarrow{a_m} (v^m, \overrightarrow{l}^m) \xrightarrow{t_m} (v^m + t_m, \overrightarrow{l}^m) \tag{7.20}$$

be a run of "length" m in $\prod_{i=1}^{n} \mathbb{A}_i$. The action labels a_k are largely ignored in UPPAAL but here they serve admirably as references to the action transitions. Assume as induction hypothesis that all runs of length shorter than m in $\prod_{i=1}^{n} \mathbb{A}_i$ map into $\prod_{i=1}^{n} \mathbb{A}'_i$. We shall abuse notation and use the same symbols $\overrightarrow{l}^k$ $(k < n)$ for "corresponding" locations in $\prod_{i=1}^{n} \mathbb{A}'_i$.

Then the part of (7.20) up to $(v^{m-1} + t_{m-1}, \overrightarrow{l}^{m-1})$ is already a run in $\prod_{i=1}^{n} \mathbb{A}'_i$.

At each stage, $k < m$, $v^k = v^{k-1} + t_{k-1}$ with the assignments of a_k applied.

Now we have to consider three possibilities for the transition:

$$(v^{m-1} + t_{m-1}, \overrightarrow{l}^{m-1}) \xrightarrow{a_m} (v^m, \overrightarrow{l}^m)$$

We could have (1): an *internal transition* of a single component $\mathbb{A}_i$, or (2): a *binary synchronisation* between two components carrying the labels $a_m!, a_m?$ or (3): a *broadcast synchronisation* between a "sending" component carrying the label $a_m!$ and "receiving components" carrying the label $a_m?$.

First, suppose a_m to be an internal transition on $\mathbb{A}_i$. Thus,

$$\forall j \neq i : l_j^m = l_j^{m-1},$$
$$v^m = v^{m-1} + t_{m-1}$$

with update assignments on $\mathbb{A}_i$-variables only.

This lifts to $\left(\prod_{1 \leq j < i} \mathbb{A}'_j\right) \parallel \mathbb{A}_i \parallel \left(\prod_{i < j \leq n} \mathbb{A}'_j\right)$ since for $j \neq i$, l_j^m "is" already in $\mathbb{A}'_j$. (Abuse of notation again: more precisely, there is a corre-

sponding location of $\mathbb{A}'_j$ which we can label l^m_j.)

Consider the time delay $(v^m, \overrightarrow{l}^m) \xrightarrow{t_m} (v^m + t_m, \overrightarrow{l}^m)$ in $\prod_{i=1}^n \mathbb{A}_i$. For $j = 1, ..., n$, Let α_j be the invariant in $\mathbb{A}_j$ at $\overrightarrow{l}^m$. Then,

$$\forall t < t_m : (v^m + t) \models \bigwedge_j \alpha_j$$

Let α'_j be the invariant in $\mathbb{A}'_j$ at l^m_j (abuse of notation again) and for arbitrary $t < t_m$ define the valuation w_t on the $\mathbb{A}_i$, $\mathbb{A}'_i$-variables as $v^m + t$ and on the $\mathbb{A}_j$, $\mathbb{A}'_j$-variables for $j \neq i$ as v^m. Then $w_t \models \alpha_i$ and also $w_t \models \bigwedge_{j \neq i} \alpha'_j$ because for $j \neq i$, $l^m_j = l^{m-1}_j$ and we can appeal to the induction hypothesis.

Now,

$$\forall t < t_m : w_t \models \alpha_i \wedge \bigwedge_{j \neq i} \alpha'_j$$

exhibits t_m as a possible time delay "transition" in $\left(\prod_{1 \leq j < i} \mathbb{A}'_j\right) \parallel \mathbb{A}_i \parallel \left(\prod_{i < j \leq n} \mathbb{A}'_j\right)$; hence by premiss (7.16), as a time delay in $\prod_{j=1}^n \mathbb{A}'_j$. It follows (the reasoning is the same as in the induction base case) that

$$\forall t < t_m : w_t \models \bigwedge_j \alpha'_j$$

In particular, $\forall t < t_m : w_t \models \alpha'_i$. So, since w_t agrees with $v^m + t$ on $\mathbb{A}_i$, $\mathbb{A}'_i$-variables, $\forall t < t_m : (v^m + t) \models \alpha'_i$.

By repetitions of this argument for all i, using all the parts of premiss (7.16), t_m is a permissible time delay in $\prod_{i=1}^n \mathbb{A}'_i$ and the segment

$$... \xrightarrow{a_n} (...) \xrightarrow{t_n} (...)$$

extends the "tail" of (7.20) to a run in $\prod_{i=1}^n \mathbb{A}'_i$.

Now consider the second possibility: a_m is a binary synchronisation of

$\prod_{i=1}^{n} \mathbb{A}_i$. In particular, there are transitions

$$l_i^{m-1} \xrightarrow{\zeta_i, a_m!, \lambda_i} l_i^m \text{ in } \mathbb{A}_i \tag{7.21}$$

$$l_j^{m-1} \xrightarrow{\zeta_j, a_m?, \lambda_j} l_j^m \text{ in } \mathbb{A}_j \tag{7.22}$$

where $(v^{m-1} + t_{m-1}) \models \zeta_i \wedge \zeta_j$ (the guards of a_m in $\mathbb{A}_i$ and $\mathbb{A}_j$) and $v^m = (v^{m-1} + t_{m-1})[\lambda_i][\lambda_j] \models \alpha_i \wedge \alpha_j$, the invariants at $l_i^m,\ l_j^n$. The semantic rules of UPPAAL are satisfied and the update assignments $\lambda_i,\ \lambda_j$ applied in the correct order.

Also, in view of the time delay $(v^m,\ \overrightarrow{l}^m) \xrightarrow{t_m} (v^m + t_m,\ \overrightarrow{l}^m)$ in $\prod_{h=1}^{n} \mathbb{A}_h$, we have $\forall t < t_m : (v^m + t) \models \alpha_i \wedge \alpha_j$.

This a_m synchronised action lifts to

$$\left(\prod_{1 \leq h < i} \mathbb{A}'_h\right) \parallel \mathbb{A}_i \parallel \left(\prod_{1 \leq i < h < j} \mathbb{A}'_h\right) \parallel \mathbb{A}_j \parallel \left(\prod_{j < h \leq n} \mathbb{A}'_h\right) \tag{7.23}$$

(without loss of generality, we may assume $i < j$) since for $h \neq i, j$, l_h^m is "already" (by abuse of notation) already in $\mathbb{A}'_h$.

As before, considering the time delay t_m in $\prod_{h=1}^{n} \mathbb{A}'_h$, where for $h = 1, ..., n$, α_h be the invariant in $\mathbb{A}_h$ at $\overrightarrow{l}^m$,

$$\forall t < t_m : (v^m + t) \models \bigwedge_h \alpha_h$$

Let α'_h be the invariant in $\mathbb{A}'_h$ at l_h^m (abuse of notation again) and for arbitrary $t < t_m$ define the valuation w_t on the $\mathbb{A}_i,\ \mathbb{A}'_i,\ \mathbb{A}_j,\ \mathbb{A}'_j$-variables as $v^m + t$ and on the $\mathbb{A}_h,\ \mathbb{A}'_h$-variables for $h \neq i, j$ as v^m. Then $w_t \models \alpha_i \wedge \alpha_j$ and also $w_t \models \bigwedge_{j \neq i,j} \alpha'_j$ because for $h \neq i, j$, $l_h^m = l_h^{m-1}$ and we can appeal to the induction hypothesis.

Now,

$$\forall t < t_m : w_t \models \alpha_i \wedge \alpha_j \wedge \bigwedge_{h \neq i,j} \alpha'_h$$

exhibits t_m as a possible time delay "transition" in the product automaton 7.23; hence by premiss (7.17), as a time delay in $\prod_{h=1}^{n} \mathbb{A}'_h$. It follows as before that

$$\forall t < t_m : w_t \models \bigwedge_h \alpha'_h$$

In particular, $\forall t < t_m : w_t \models \alpha'_i \wedge \alpha'_j$. So, since w_t agrees with $v^m + t$ on $\mathbb{A}_i,\ \mathbb{A}'_i,\ \mathbb{A}_j,\ \mathbb{A}'_j$-variables, $\forall t < t_m : (v^m + t) \models \alpha'_i \wedge \alpha'_j$.

t_m is a permissible time delay in $\prod_{i=1}^{n} \mathbb{A}'_i$ and the segment

$$... \xrightarrow{a_n} (...) \xrightarrow{t_n} (...)$$

extends the "tail" of (7.20) to a run in $\prod_{i=1}^{n} \mathbb{A}'_i$.

The third possibility is that a_m is broadcast synchronisation. The computational details of the inductive step are in this case similar to the previous one, except instead of a single receiving edge in component j there are several, in components $j_1, ..., j_r$ and so the "exceptional" edges are in $i, j_1, ..., j_r$.

Thus the run (7.20) is in $\prod_{h=0}^{n} \mathbb{A}'_h$ in the case of a synchronised transition and the induction is complete.

7.6 Further Composition Theorems

In applications, it is useful to replace one component of a system with an *abstraction* with the same external interface: synchronising actions (at least) and locations, and have in consequence a system which simulates the original.

7.6.1 Definition

Given two timed automata $\mathbb{A} = (L, l^0, A, \mathcal{H}, E, I)$ and
$\mathbb{A}' = (L', l'^0, A, \mathcal{H}, E', I')$ with the same set of clocks and set of action labels, call a relation $R \subseteq L \times L'$ a *component-simulation* if for all $(l, l') \in R$, for

every edge from l, $l \xrightarrow{\zeta,a,\lambda} m$ there exists an edge from l', $l' \xrightarrow{\zeta',a,\lambda'} m'$[5], with

- the same action label;
- targets related by R: $(m, m') \in R$;
- $\zeta \vdash \zeta'$, $I(m) \vdash I'(m')$ and $\lambda' = \lambda$.

If $\mathbb{A}$, $\mathbb{A}'$ are UPPAAL automata (process templates), this definition applies with $a \in A$ understood as an "empty" ("anonymous", "τ") action label or a synchronisation label, and λ, λ' as sets of the more general UPPAAL assignment/updates involving variables of UPPAAL types, not just clocks.

7.6.2 Proposition

Let let $\mathbb{A}$, $\mathbb{A}'$ be automata as above and suppose them to be components of composite system: $\mathbb{A} \parallel ...$ and $\mathbb{A}' \parallel ...$, the ellipses representing the other components which are the same in two systems. Let $\mathcal{V}$ denote the set of *valuations* of clocks and other variables (interpreted in a suitably general way in the case of UPPAAL automata) of the two systems. Given a component-simulation $R \subseteq L \times L'$, define a relation $\hat{R}$ between timed states of the systems by
$(l, ..., v; l', ..., v,) \in \hat{R}$ iff $(l, l') \in R$ and the remaining component locations match, and $v = v'$ in $\mathcal{V}$.
Then $\hat{R}$ is a simulation of timed transition systems 2.4.2.

Proof. Say $(l, l') \in R$, $v = v'$ and there is a timed transition $(l, ..., v) \xrightarrow{a} (m, ..., w)$ in the $\mathbb{A}$-based system. This must arise from some edge (which may or may not be participating in a synchronised action) $l \xrightarrow{\zeta,a,\lambda} m$ of $\mathbb{A}$, with $v \models \zeta$ and $w = v[\lambda] \models I(m)$. ($v[\lambda]$ means v with the updates – assignments and clock resets of λ.)

By hypothesis, there exists an edge with the same label, $l' \xrightarrow{\zeta',a,\lambda'} m'$ with $(m, m') \in R$, $\zeta \vdash \zeta'$, $I(m) \vdash I'(m')$ and $\lambda' = \lambda$: so $v' = v \models \zeta'$ and $w' \triangleq w = v[\lambda] = v'[\lambda'] \models I'(m')$. There is therefore a corresponding timed

[5]The $\longrightarrow$ refers to E or E' according to context.

transition in the $\mathbb{A}'$-based system: $(l', .., v') \xrightarrow{a} (m', ..., w')$. If synchronisation is required, the requirements carry over from the one system to the other.

7.6.3 Definition

Given two timed automata $\mathbb{A} = (L, l^0, A, \mathcal{H}, E, I)$ and
$\mathbb{A}' = (L', l'^0, A, \mathcal{H}, E', I')$ with the same set of clocks and set of action labels, call a relation $B \subseteq L \times L'$ a *component-bisimulation* if for all $(l, l') \in R$,

- For every edge from l, $l \xrightarrow{\zeta,a,\lambda} m$ there exists an edge from l', $l' \xrightarrow{\zeta',a,\lambda'} m'$, with the same action label, targets related by R: $(m, m') \in R$, and $\zeta \vdash \zeta'$, $I(m) \vdash I'(m')$ and $\lambda' = \lambda$.
- For every edge from l', $l' \xrightarrow{\zeta',a,\lambda'} m'$ there exists an edge from l, $l \xrightarrow{\zeta,a,\lambda} m$, with the same action label, targets related by R: $(m, m') \in R$, and $\zeta' \vdash \zeta$, $I'(m') \vdash I(m)$ and $\lambda = \lambda'$.

7.6.4 Proposition

Let let $\mathbb{A}$, $\mathbb{A}'$ be components of composite systems as above, the other components being the same in the two systems. Let $\mathcal{V}$ denote the set of *valuations* of clocks and other variables . Given a component-simulation $B \subseteq L \times L'$, define a relation $\hat{B}$ between timed states of the systems by
$(l, ..., v; l', ..., v,) \in \hat{B}$ iff $(l, l') \in R$ and the remaining component locations match, and $v = v'$ in $\mathcal{V}$.
Then $\hat{B}$ is a strong bisimulation of timed transition systems 2.4.1.

The proof is is essentially a repetition of the previous proof in two logical directions.

7.7 Conclusions

7.7.1 Hybrid Automata, UPPAAL

The formal proof given in subsection 7.4.2 dealt specifically with *timed automata* but most tool sets and many modelling situations deal with more

general *hybrid* automata. These carry an arbitrary set of variables, not just clocks, and a subset of variables can be updated to arbitrary values (not just reset to 0) on each discrete action. Actually, hybrid automata come in many flavours, some featuring real (continuous) variables, some integers only, some, like the UPPAAL toolset [109], quite a complex type system including arrays. Since UPPAAL is such a useful tool for applying the results of this thesis, developing compositionality results for UPPAAL automata seems a good idea.

7.7.2 The $\preceq$ Relation

The $\preceq$ relation between automata corresponds loosely to the notion of satisfaction of specified properties of a system. Behaviour can be *specified* by an automaton $\mathbb{S}$; then $\mathbb{A} \preceq \mathbb{S}$ tells us that the behaviour described by $\mathbb{A}$ *implements* the specification $\mathbb{S}$.

The UPPAAL tool set includes a CTL-like language (including modal operators $\Box$ = "always", $\Diamond$ = "eventually", $\mathcal{U}$ = "until", and $\forall, \exists$-quantification over execution paths) described in [109] for expressing properties of an automaton or, more generally, of systems modelable by automata.

It will useful that, whenever $\mathbb{A} \preceq \mathbb{S}$ and φ is formula of the specification language expressing some system property, then $\mathbb{S} \models \varphi$ implies $\mathbb{A} \models \varphi$. In fact this is the case provided φ employs only *universal* quantification over paths: see, for instance Clarke *et al.* [45], p177, Theorem 16. Briefly, if φ is $\forall\Box\psi$, satisfied by $\mathbb{S}$, then ψ is true at every state on every run of $\mathbb{S}$, hence *a fortiori* at every state on every run of $\mathbb{A}$. Similarly if φ is $\forall\Diamond\psi$: ψ is true at*some* state on every run of $\mathbb{S}$, hence at some state on every run of $\mathbb{A}$.

A related question is, given a property φ for which we would like to know whether some model $\mathbb{A} \models \varphi$, whether there is some more abstract specification $\mathbb{S}$ such that $\mathbb{A} \preceq \mathbb{S} \models \varphi$.

7.7.3 Decomposition

The case studies to be considered in the sequel make use of UPPAAL which does in fact support hybrid automata, although in the first instance we are

primarily interested in modelling and checking bCANdle models via the plain timed-automaton representation. We shall in the sequel show that UPPAAL semantics *simulates* timed-automata semantics in the sense of 2.4.2.

So far, work in UPPAAL with questions such as those of the previous subsection has run frustratingly often into the state-explosion problem: it will be useful, therefore, to decompose automaton-based models into parallel-composites and employ results such as those obtained in this chapter. The work of the next chapter will be to examine some examples of how this may be done in practice, and the benefits that accrue from this approach.

Chapter 8

Compositional Model Checking in Practice

In this chapter, examples will show how the results of the last chapter may be used with decomposition to make checks of composite models faster, using less memory, and, indeed, to make previously intractable checks tractable. The focus is on distributed timed systems in which components communicate via CAN, modelled in UPPAAL using the CAN representation described in chapter 6. The checks were performed using the UPPAAL verifier tool.

8.1 A Translation Tool

There is a practical problem with the no-shared-variables CAN model. For a given value of the parameters m and n, the number of message types and values (of each type) supported, the UPPAAL source hard-codes multiple synchronisation "channels", locations and edges with multiplicity m or $m \times n$ (denoted by the thick lines in figure 6.1 and by the indices μ, ν in this figure and in the pseudo code at the end of §6.1). In early experiments this was coded by hand, and quickly became tedious.

To help with this, `CANGen`, a code-generating "front-end" to UPPAAL was developed, which reads an input file and generates UPPAAL source code in `.xta` format. The input file starts with lines specifying values of m and n;

then a line beginning `#channel` specifies that code for a CAN channel be generated, using `l,u,L,U`-parameters given on the line.

After this are lines of ordinary UPPAAL source code interspersed with other CANGen directives

- `#fanoutTypes` generates code for a process to fan an urgent synchronisation out over multiple channels;
- `#globals` causes subsequent input to be placed in the global declarations area of the UPPAAL source;
- `#processes` causes subsequent input to be placed in the process template area of the UPPAAL source;
- `#procInstances` causes subsequent input to be placed in the process instance definition area of the UPPAAL source;
- `#sysDef` causes subsequent input to be placed in the system definition area of the UPPAAL source.

The most important syntactic constructs in the input to `CANGen` are

- `FORType(t)` ... *token*`|t|`... `ROF`
- `FORVal(v)` ... *token*`|v|`... `ROF`

These cause input tokens to be replicated over all message idetifiers (types) and, respectively, all message values. The sequence of tokens between `FORType(t)` and `ROF` is replicated m times, each instance of `|t|` becoming `000` then `001` and so on up to $m-1$. The sequence of tokens between `FORVal(v)` and `ROF` is replicated n times. There may be multiple occurrences of `|t|` or `|v|` in a sequence being replicated; the metavariable may be something other than `t` or `v` but must match the one in the `FOR` token.

The replicated sequences are generated as a list separated by "," or by ";" depending on the UPPAAL context.

The construct may run over several lines in the source file: `\n` is just a source code character as far as `CANGen` is concerned.

These constructs may be nested. For instance,

```
FORtype(t) FORval(k) broadcast chan &rdg|t|_|k| ROF ROF
```

expands in the case $m = 2, n = 3$ to

```
broadcast chan &rdg000_000, broadcast chan &rdg000_001,
broadcast chan &rdg000_002, broadcast chan &rdg001_000,
broadcast chan &rdg001_001, broadcast chan &rdg001_002
```

As an illustration, the input file to `CANGen` and the generated UPPAAL source code are listed in appendix D.

8.2 A First Example

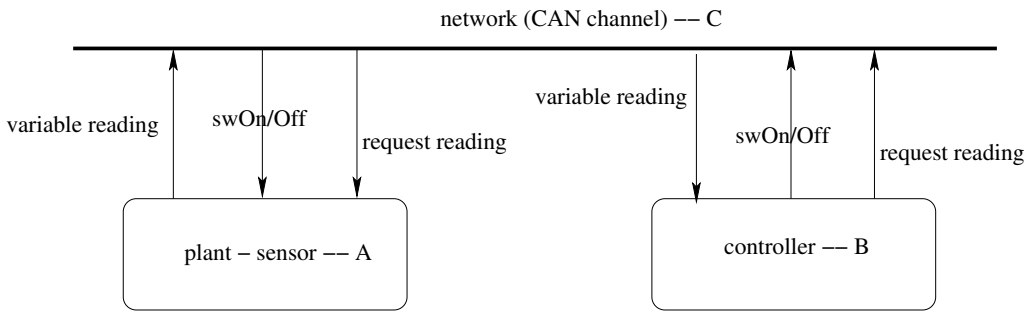


Figure 8.1: A distributed control system: the component processes and messages

Figure 8.1 shows a small control system comprising a sensing process and a controller process communicating via a CAN network.

The sensing process $\mathbb{A}$ might naturally be conceived as a composite, a "plant", $\mathbb{P}$ synchronising in parallel with a "sensor" process $\mathbb{S}$ as in figure 8.2. After some analysis one may obtain a simulation $\mathbb{P} \parallel \mathbb{S} \preceq \mathbb{A}$, the latter being a combined plant-sensor process like the one depicted in figure 8.3.

The controller $\mathbb{B}$ is depicted in figure 8.4 and the connecting CAN network is represented by a channel process $\mathbb{C}$. From proposition 7.6.2 we can infer that $\mathbb{P} \parallel \mathbb{S} \parallel \mathbb{B} \parallel \mathbb{C} \preceq \mathbb{A} \parallel \mathbb{B} \parallel \mathbb{C}$ and so the system is modelled at an abstract level as $\mathbb{A} \parallel \mathbb{B} \parallel \mathbb{C}$. The `CANGen` source code for this is to be found in appendix

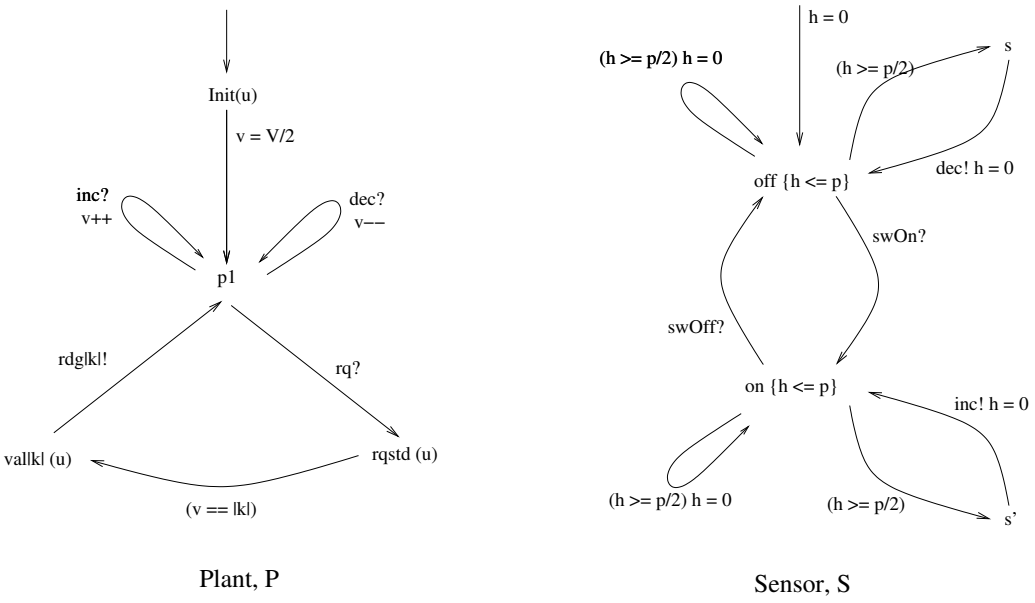


Figure 8.2: Synchronising Plant and Sensor processes

E.1. The system can be in an "on" state or an "off" state and features a variable which non-deterministically increments when the state is "on", and decrements when the state is "off". The variable, perhaps a temperature, is being continuously monitored by the sensor process and readings fed through a network to the controller process which decides whether to switch the system state (perhaps a heater) on or off.

The sensing process (see figure 8.3) has a duty cycle in which it repeatedly waits for a request (delivered through the network, denoted `rq`) from the controller for a sensor reading, then outputs the current reading onto a network. While waiting (at location `p1`), it simulates the variable being incremented or decremented non-deterministically, and at any location in the duty cycle it can respond to a request `swOn` or `swOff` from the controller via the network, to switch the system state (heater) on or off. This is, perhaps, more than simply a sensing process as it simulates the variable being sensed as well.

`CANGen` is modelling the network for the system using three message types and `V` distinct values. Type 0 messages from the plant-sensor deliver readings

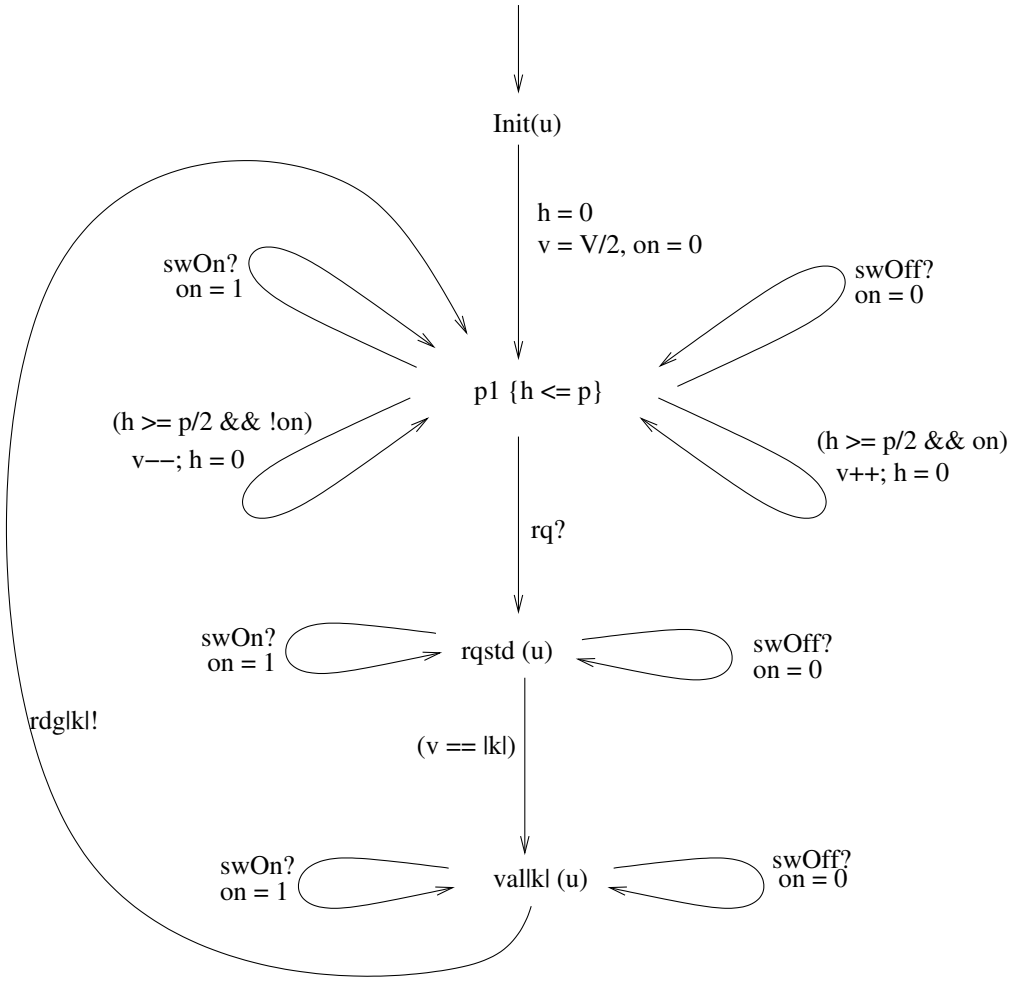


Figure 8.3: A Plant Sensor process with a single sensor

to the controller; type 1 messages deliver switch-on, switch-off (values 1, 0) requests from the controller, and type 2 messages reading requests. In figure 8.3, `p` denotes an integer parameter to the plant sensor process template, and `g`, `h` denote UPPAAL clock variables.

The controller process (figure 8.4) duty cycle begins at location `rqstVal` and repeatedly requests a reading by issuing the `rq!` synchronisation and then obtains a sensor reading from the network, assigning the value to variable `vAv`. Parameters `lo`, `hi` define limits within which the controller would like to keep the sensor reading. If `vAv` falls below `lo`, an instruction to the

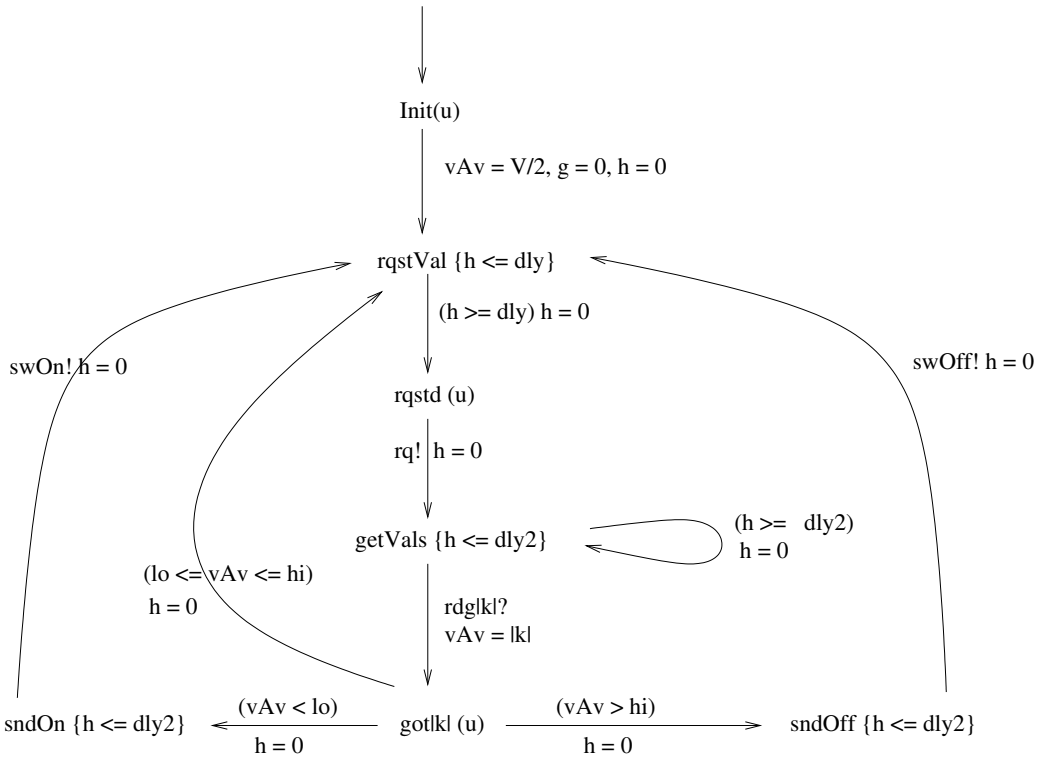


Figure 8.4: A Plant Controller process

sensing process to "switch on" is issued; if `vAv` rises above `hi`, an instruction to "switch off" is issued; otherwise, the process returns to the beginning of its duty cycle.

With this system, one can check various desirable properties such as that it will not deadlock, and that values of the variable are kept within bounds. Properties such as these are expressible in the UPPAAL specification language (essentially CTL):

- $\neg\exists\Diamond$deadlock (or $\forall\Box\neg$deadlock)
- $\forall\Box$(controller.vAv < 28)
- $\forall\Box$(controller.getVals $\Rightarrow$ controller.vAv > 12)

When `lo` and `hi` are given as 19 and 21 respectively, UPPAAL's `verifyta`

tool verifies[1] that these three properties are satisfied. Further, verifyta can be run inside J Bengtsson's `memtime` tool to obtain execution times and measures of memory usage. This simple software tool, obtainable from the UPPAAL web site, forks a child process within which the command (e.g. `verifyta`) runs, while the parent process gathers time and memory usage statistics for the child process from its `/proc/`(*pid*)`/stat` file. CPU time (in user and in kernel mode) and maximum (virtual) memory allocated to the process and maximum resident set size (real memory usage – the size of segment of the process permanently resident in memory) are reported.

Results for the present case, obtained on a simple Intel-based machine,[2] are as below. These figures are averages over 20 trials: means and stahdard deviations are quoted

$\forall\Box$(controller.getVals $\Rightarrow$ controller.vAv $> \xi$) is abbreviated $o(\xi)$ and $\forall\Box$(controller.vAv $< \xi$) is abbreviated $u(\xi)$. Maximum memory usages (virtual memory usage and resident set size) are reported in megabytes.

check:	$o(12)$	$o(14)$	$o(16)$	$u(24)$	$u(26)$	$u(28)$
mean user time:	94.0”	96.5”	92.9”	94.6”	94.9”	93.3”
standard dev :	0.8	1.0	0.8	1.6	2.3	0.5
mean max v mem:	43.331	43.331	43.332	43.330	43.330	43.330
standard dev :	0.003	0.003	0.003	0.002	0.002	0.003
mean max rss:	8.851	8.851	8.851	8.851	8.849	8.849
standard dev :	0.004	0.003	0.003	0.004	0.004	0.002

A check that the system is deadlock-free took somewhat longer, 693.1 seconds averaged over 20 trials, with stamdard deviation 13.8; maximum virtual memory usage was 45.571 Mb (SD = 0.003) and maximum rss was 11.5 Mb (SD = 0.003).

It would be good to see whether such desirable behaviour of the system is also guaranteed in more “developed” versions of this control system, and these figures provide a base line for comparison with the computing effort associated with this.

[1] using breadth-first search and conservative state space optimisation

[2] an Acer Aspire 1360 running at 2.8 GHz with 1Gb RAM, running Ubuntu 8.04

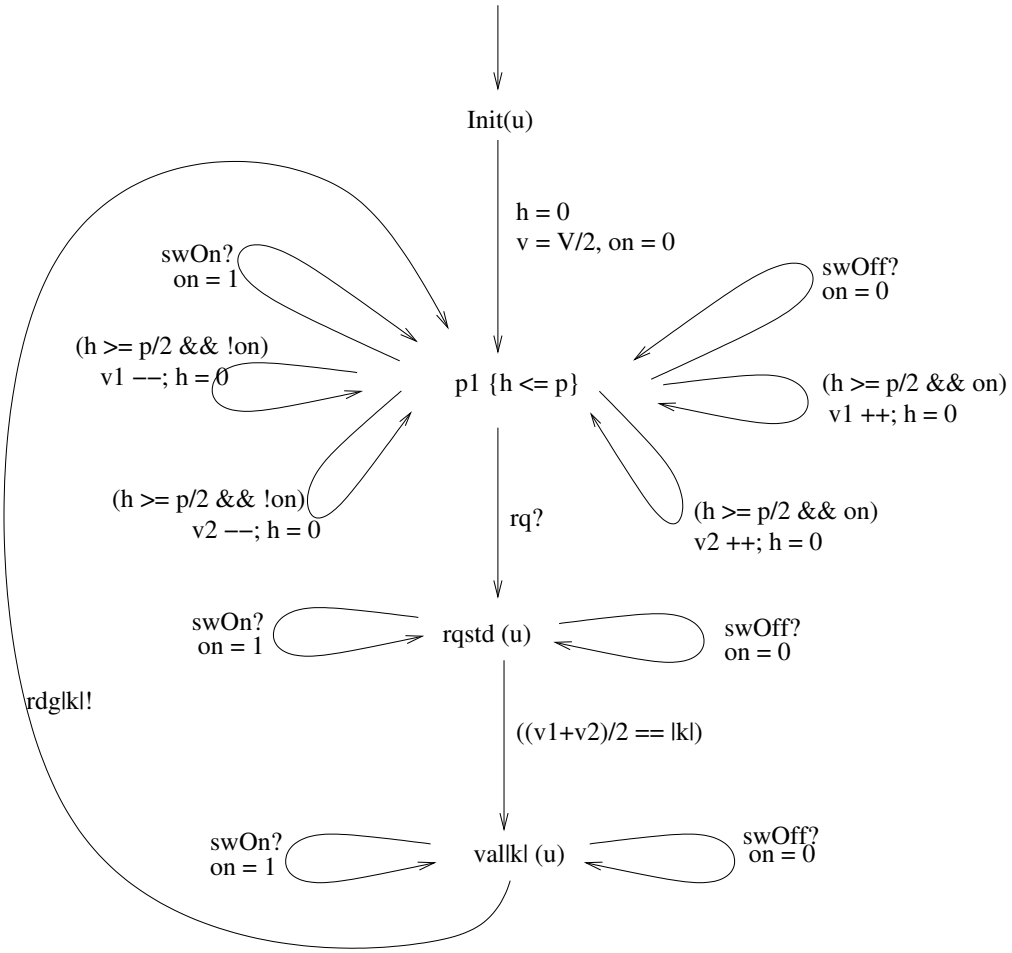


Figure 8.5: A Plant Sensor process with two sensors, averaged

For instance, one might decide to use two sensors whose readings are averaged instead of a single sensor. Figure 8.5 shows a replacement for the sensing process in which *two* variables are sensed (again this is simulated by non-deterministic random-walk) and on request, their *average* is output to the network for the controller to pick up. The controller process is the same as before.

With just this admittedly artificial but simple increase in complexity, it now takes UPPAAL a long time (over an hour) to verify any of the $o(\xi)$, $u(\xi)$ properties. For instance a check of $o(12)$ inside memtime took 1 hour, 44 minutes, 33 seconds. Total memory usage in this case rose to 189 Mb (rss to

155 Mb) – still feasible with current resources.

However with a further increase in complexity, to a system which averages *three* sensors, verfication of $o(\xi)$ failed altogether. After a little over 10 hours, with memtime reporting 10h 3m 50s elapsed CPU time, memory usage rose dramatically to 936 Mb (around the limit of installed RAM) and the disk drive started thashing. Therafter, the CPU time indication made very little progress. Memory demand (the size of the state space) had outstripped the machine.

Clearly this limit could be pushed further out by using a bigger machine; but then what about a further small increase in the complexity of the problem? Fortunately, all the properties considered here can be *inferred* from the corresponding properties of the simpler 1-sensor system. First, if one imagines computing `(v1+v2)/2` *every* time `v1` or `v2` is incremented or decremented by the plant sensor process at location `p1`, then the externally visible behaviour of the two-sensor process is (with this addition) just that of a state machine looking like that of the one-sensor process (8.3) except with (for all k) the transition `rqstd` $\longrightarrow$ `val|k|` guarded by `(v/2 == |k|)`.

This in turn is easily seen to be simulated by a version of the one-sensor process. The simulation relation is the diagonal. One may now use the proposition 7.6.2 to *infer* the properties set out above for the two-sensor system: it is simulated by a one-sensor system for which corresponding properties are easily verified; and for any execution of the the simulated system there is a corresponding execution of the simulating system with the desired properties.

The reasoning behind this example generalises to systems which average more than two sensors, for which a direct check is, as we have seen, intractable.

8.3 Using Assume-Guarantee

8.3.1 Another Control System

Consider now a control system with the same high-level structure as the examples of the previous section (Figure 8.6).

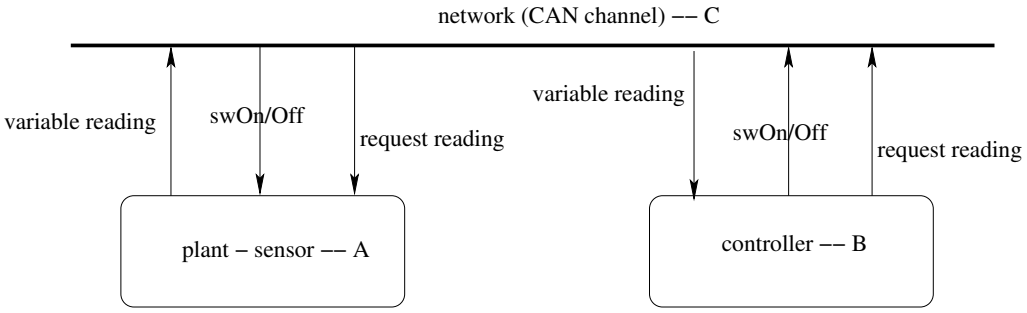


Figure 8.6: Distributed control system: the component processes and messages

A plant-sensor process models and simulates a plant which obtains readings of a sensed variable and sends them through a CAN network to a controller process. The controller sends requests for readings, receives these readings and sends switch-on, switch-off requests to the plant-sensor process. The plant-sensor process non-deterministically increments the sensed variable when it is "on" and decrements it when "off". So far, this is very like the examples of §8.2.

Imagine now, though, that we wish to model a kind of system inertia in the plant-sensor process. In particular, there is an "inertia" parameter which specifies a minimum time that must elapse from a switch-on or switch-off before the change of state takes effect. Before this time has elapsed, the variable does not change value.

There is also some inertia now in the controller. In the previous version of this, there was a "switch-on threshold" and "switch-off threshold"; the controller could issue a switch-on command if a received value was below the switch-on threshold, and a switch-off command if it was above the switch-off threshold. A more complex logic is now modelled. There is, in addition, a

parameter e, and also an "inertia" parameter provided. When a reading is received which is below the lower threshold or above the upper threshold, a switch-on or switch-off command (as appropriate) may be issued *provided* a time of at least the inertia parameter has elapsed since the last issue of a switch-on or -off. A switch-on or -off will also be issued if the received reading is below the lower or above the upper threshold by e or more, regardless of the time elapsed. Otherwise no switch-on or -off is issued. This models a situation where the controller may respond sluggishly to small crossings of the thresholds, but respond urgently to large deviations (modelled by parameter e).

As before the system is modelled as an UPPAAL system $\mathbb{A} \parallel \mathbb{B} \parallel \mathbb{C}$ composed of three processes: ($\mathbb{A}$) the plant-sensor process (figure 8.7), ($\mathbb{B}$) the controller process (figure 8.8), and ($\mathbb{C}$) the CAN network channel.

The plant-sensor process sends readings of its variable v through the network channel as before (as messages of type `000`) to the controller process. The controller sends commands to "switch on" or "switch off" to the plant-sensor as before, as messages of type `001`, value `000` for "off", value `001` = "on"; and requests for a reading as messages of type `002`.

As before, the plant-sensor process simulates a variable `v` whose value, initially set to the middle of the range of integer values modelled by the channel process, goes up when the system is "on" and down when it is "off". After half the permitted delay at location `p1`, `v` *may* (but does not have to) increment if the system is "on" and decrement if it is "off", provided also that the required "inertia time" has elapsed. The latter is modelled by the conjunct `g > in` in the guards of the self-loops at `p1` which increment and decrement `v`. Clock `g` is reset whenever a request to switch on or switch off is received. This may happen at every location in the duty cycle. Bear in mind that these messages are, in the fashion of UPPAAL broadcast-synchronisation semantics, picked up as soon as they are available.

Similarly, the plant-sensor process may not linger at `p1` when a `rq` message arrives, but must proceed to the rest of the duty cycle in which a reading of `v` is sent through the network to the controller in a fashion similar to the the example of §8.2. An UPPAAL urgent synchronisation `rdg|k|!` sends

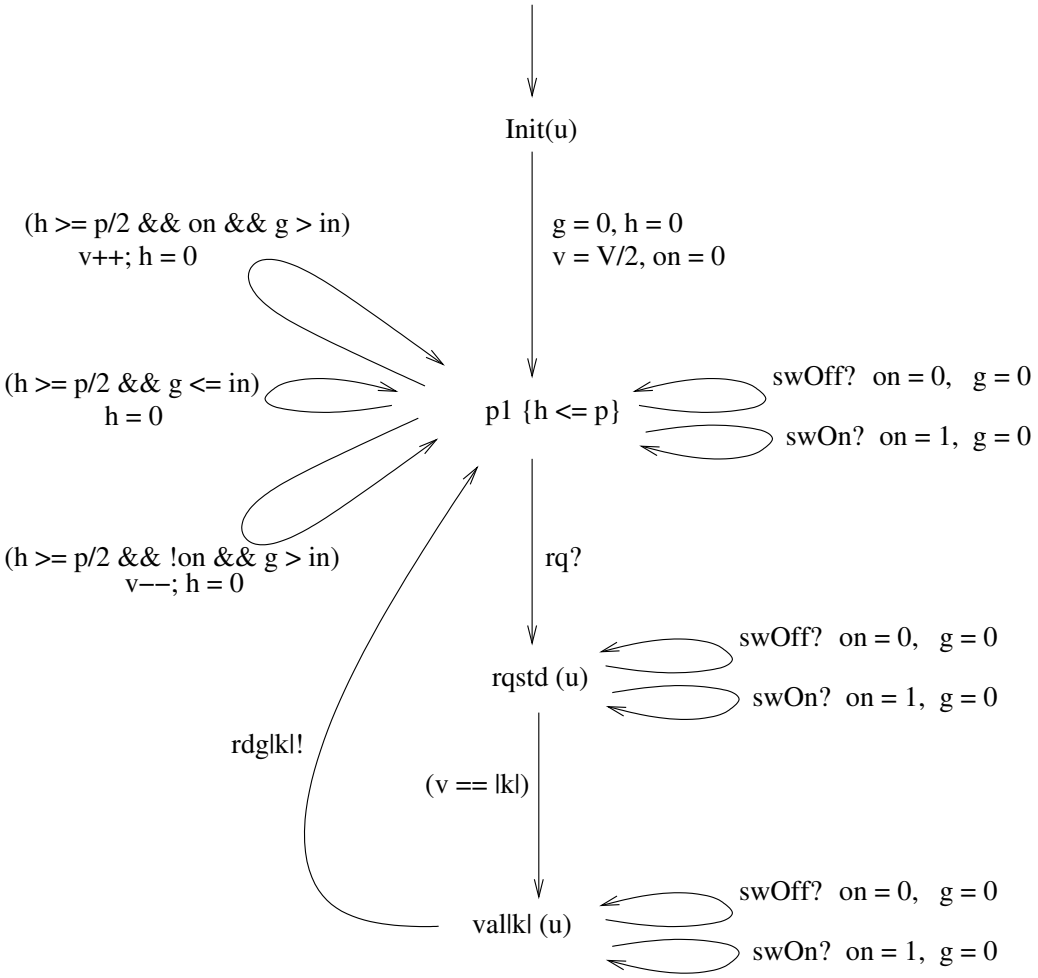


Figure 8.7: A plant-sensor process

the value `k` onto the network where it is eventually received by the controller process.

The controller process $\mathbb{B}$ is shown in figure 8.8. In addition to the features of the previous version, there is a clock `g` measuring time since the last switch-on or switch-off command was sent. Guards test clock `g` against an "extra inertia" parameter in the process of deciding whether to issue switch-on or switch-off instructions.

Its duty cycle begins at location `rqstVal` with, after a delay, a move to a location `rqstd` from which a request to the plant-sensor process for a variable

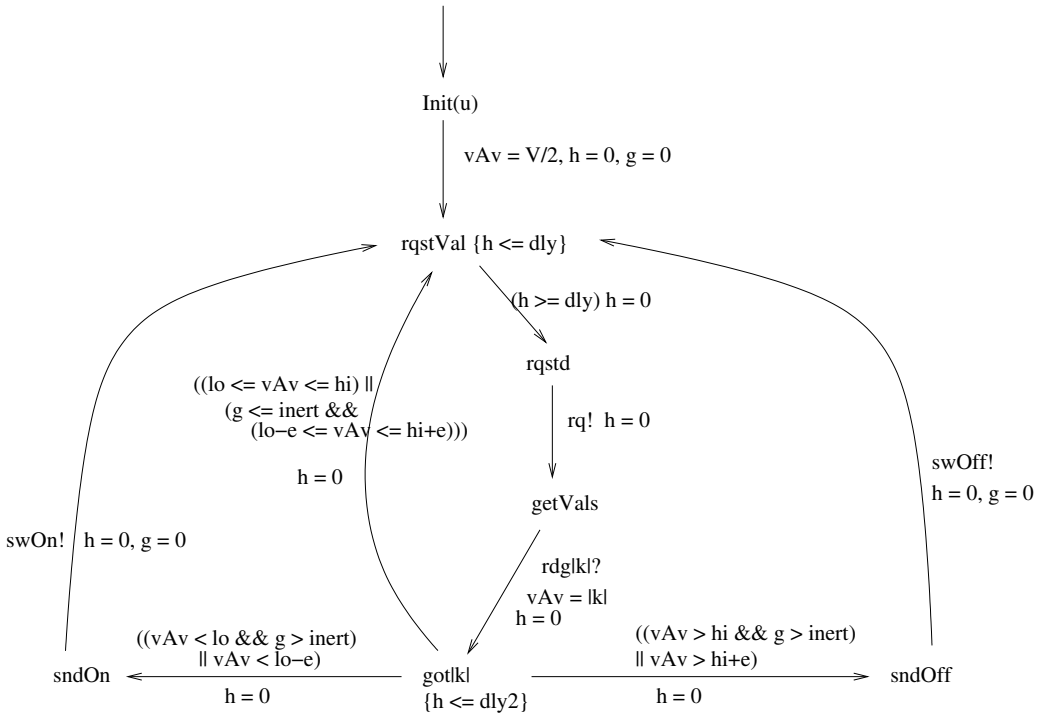


Figure 8.8: A controller process

reading is sent to the network. This happens by means of an UPPAAL urgent synchronisation, as soon as it can happen. Then, from location `getVals`, a value is received from the network and assigned to the variable `vAv`.

Bear in mind that as usual in CANGen notation, `got|k|` is actually shorthand for a set of locations (`k` = 0 ... 39) and the edges into and out of `got|k|` are likewise sets of edges. The edges from `getVals` "fan out" and the edges to `sndOn` "fan in", as do the edges to `sndOff` and back to `rqstVal`.

If *either* the value of `vAv` has fallen below `lo` and time since the last switch-off or -on (measured by `g`) exceeds `inert`, *or* `vAv` has fallen below `lo - e`, the edge to `sndOn` *may* be followed, whence a switch-on message is urgently sent to the network (and thence to the plant-sensor). If *either* `vAv` has risen above `hi` and `g > inert`, *or* `vAv` has risen above `hi + e`, the edge to `sndOff` *may* be followed, whence a switch-off message is sent. However, as long as the value of `vAv` is *between* `lo-e` and `hi+e` and less time than `inert` has elapsed since the last switch-on or -off, the edge back to`rqstVal` may be

taken, bypassing the sending of switch-on or switch-off messages.

The CANGen source code for this model is listed in appendix F.1.

Can this model deadlock? How good is the control provided by it? In experiments, the "nominal" value of the variable `plantSensor.v` was set to 20: `plantSensor.v` and `controller.vAv` were initialised to this. `controller.lo` and `controller.hi` are 19, 21 respectively. The properties

$$[o(\xi)]: \ \forall\Box(\text{controller.getVals} \Rightarrow \text{controller.vAv} > \xi)$$

were checked for $\xi = 12, 14, 16$ and the properties

$$[u(\xi]): \ \forall\Box(\text{controller.vAv} < \xi)$$

were checked for $\xi = 24, 26, 28$. As in the previous section, the checks were by UPPAAL `verifyta` running inside `memtime`. The results are means and standard deviations of 20 trials. When `e` was set at 2, all these properties were satisfied. Here are the times and memory usage of these checks when e = 2:

check:	$o(12)$	$o(14)$	$o(16)$	$u(24)$	$u(26)$	$u(28)$
mean user time:	261.2"	264.9"	252.2"	252.6"	252.8"	256.9"
standard dev :	5.6	9.1	0.5	0.5	1.0	7.0
mean max v mem:	43.415	43.415	43.417	43.415	43.415	43.415
standard dev :	0.003	0.002	0.003	0.002	0.002	0.003
mean max rss:	9.002	8.851	8.999	8.998	8.999	9.001
standard dev :	0.008	0.009	0.005	0.006	0.006	0.007

In all cases the model was found also to be deadlock-free: $\neg\exists\Diamond$deadlock (or $\forall\Box\neg$deadlock). However, the check for this took much longer: the companion to the result above took 2461 seconds (41 minutes, 1 second) averaged over 20 trials, with a standard deviation of 40 seconds; maximum (virtual) memory reached 45.4 Mb and rss 11.5 Mb, on average with standard deviations of 0.003 Mb, 0.004 Mb respectively.

It would be useful to these properties could be checked more rapidly. This

can in fact be done if the component $\mathbb{A}$ can be replaced by a simpler, faster to check, component $\mathbb{A}'$ which nevertheless *simulates* it (in the presence of the rest of the system), and if the component $\mathbb{B}$ is likewise replaced by a simpler abstraction $\mathbb{B}'$. For this to be valid, we need to know that all timed runs of $\mathbb{A} \parallel \mathbb{B} \parallel \mathbb{C}$ are also timed runs of $\mathbb{A}' \parallel \mathbb{B}' \parallel \mathbb{C}$. To be useful, the abstract model must be quicker to check.

We shall construct models $\mathbb{A}', \mathbb{B}'$ by *abstracting* from $\mathbb{A}, \mathbb{B}$ and find that the checks can indeed be performed much more quickly on $\mathbb{A}' \parallel \mathbb{B}' \parallel \mathbb{C}$ which the model $\mathbb{A} \parallel \mathbb{B} \parallel \mathbb{C}$ *simulates*. The relation $\mathbb{A} \parallel \mathbb{B} \parallel \mathbb{C} \preceq \mathbb{A}' \parallel \mathbb{B}' \parallel \mathbb{C}$ is inferred using the assume-guarantee theorem 7.5.2 and, using this, the $o(\xi)$, $u(\xi)$ and deadlock-freeness properties for $\mathbb{A} \parallel \mathbb{B} \parallel \mathbb{C}$ can be inferred from the analogous properties of $\mathbb{A}' \parallel \mathbb{B}' \parallel \mathbb{C}$.

$\mathbb{A}'$ is devised by abstracting out some of the logic of $\mathbb{A}$. Figure 8.9 shows the (non-deterministic) abstraction $\mathbb{A}'$ of the plant-sensor process $\mathbb{A}$ of figure 8.7. This has the same structure as $\mathbb{A}$, the only difference being that the guards on three of the loops at location `p1` are weaker. The clock `g` has disappeared; it is not needed.

Similarly, $\mathbb{B}'$ is abstracted from $\mathbb{B}$. Figure 8.10 shows the abstraction $\mathbb{B}'$ of the controller process $\mathbb{B}$ of figure 8.8. Again the structure of locations and guards is the same, but the inertia modelling involving tests on clock `g` have been replaced by weaker guards. The CANGen source code for the whole abstraction is listed in appendix F.2.

It will be seen that the properties of the system $\mathbb{A}' \parallel \mathbb{B}' \parallel \mathbb{C}$ are checkable very quickly. To be useful, though, it needs be established that runs of this system subsume those of the original system $\mathbb{A} \parallel \mathbb{B} \parallel \mathbb{C}$.

The assume-guarantee theorem 7.5.2 (page 152) is now used to establish

$$\mathbb{A} \parallel \mathbb{B} \parallel \mathbb{C} \preceq \mathbb{A}' \parallel \mathbb{B}' \parallel \mathbb{C}.$$

Indeed, given that the same version, $\mathbb{C}$ appears on both sides of this relation, the premisses (a) and (b) of the theorem are covered provided we

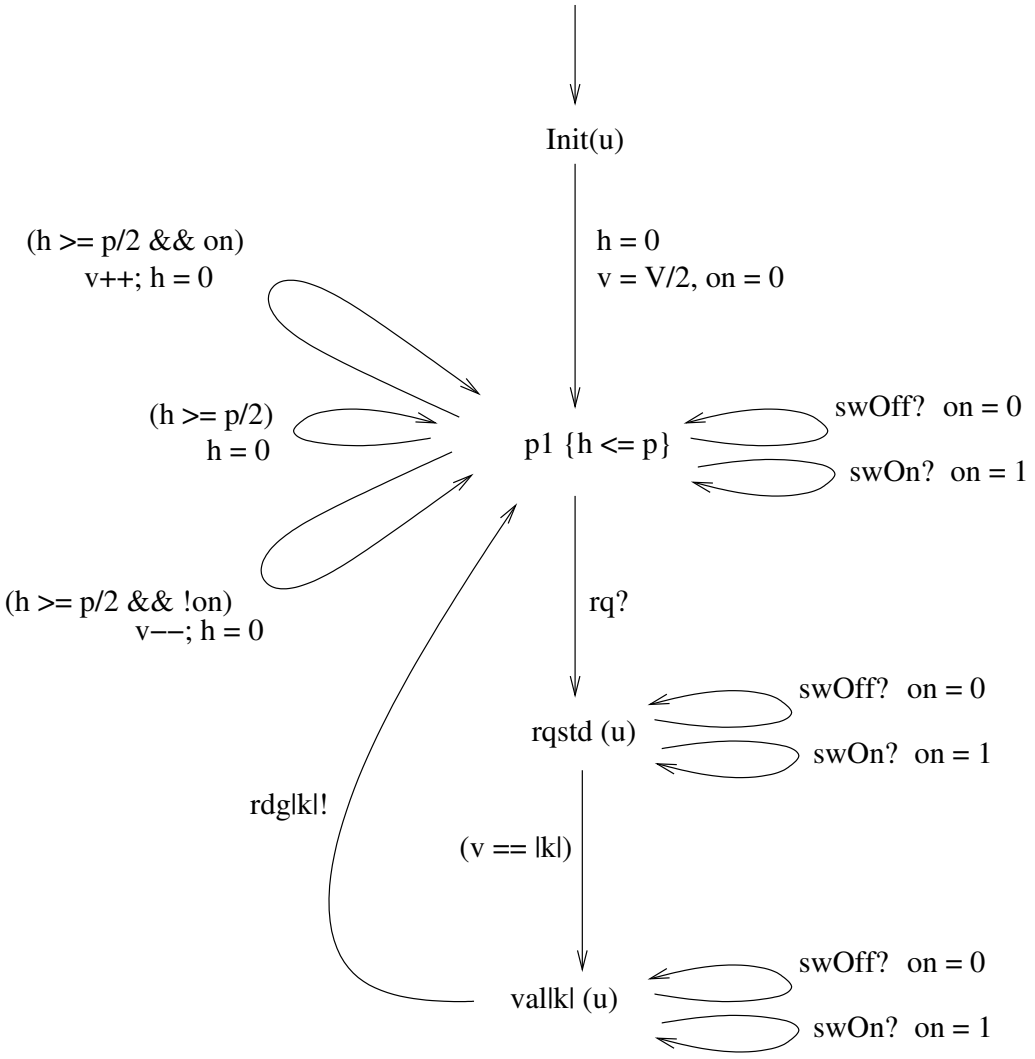


Figure 8.9: An abstract plant-sensor process

can show

$$\mathbb{A} \parallel \mathbb{B} \parallel \mathbb{C} \preceq \mathbb{A}' \parallel \mathbb{B} \parallel \mathbb{C} \quad \text{and} \quad \mathbb{A} \parallel \mathbb{B} \parallel \mathbb{C} \preceq \mathbb{A} \parallel \mathbb{B}' \parallel \mathbb{C}.$$

Now, $\mathbb{A}$ and $\mathbb{A}'$ have the same locations and transition edges, save that some guards in $\mathbb{A}$ are stronger than the corresponding guards in $\mathbb{A}'$. From this it is straightforward to infer that the diagonal relation between states provides a simulation from $\mathbb{A} \parallel \mathbb{B} \parallel \mathbb{C}$ to $\mathbb{A}' \parallel \mathbb{B} \parallel \mathbb{C}$.

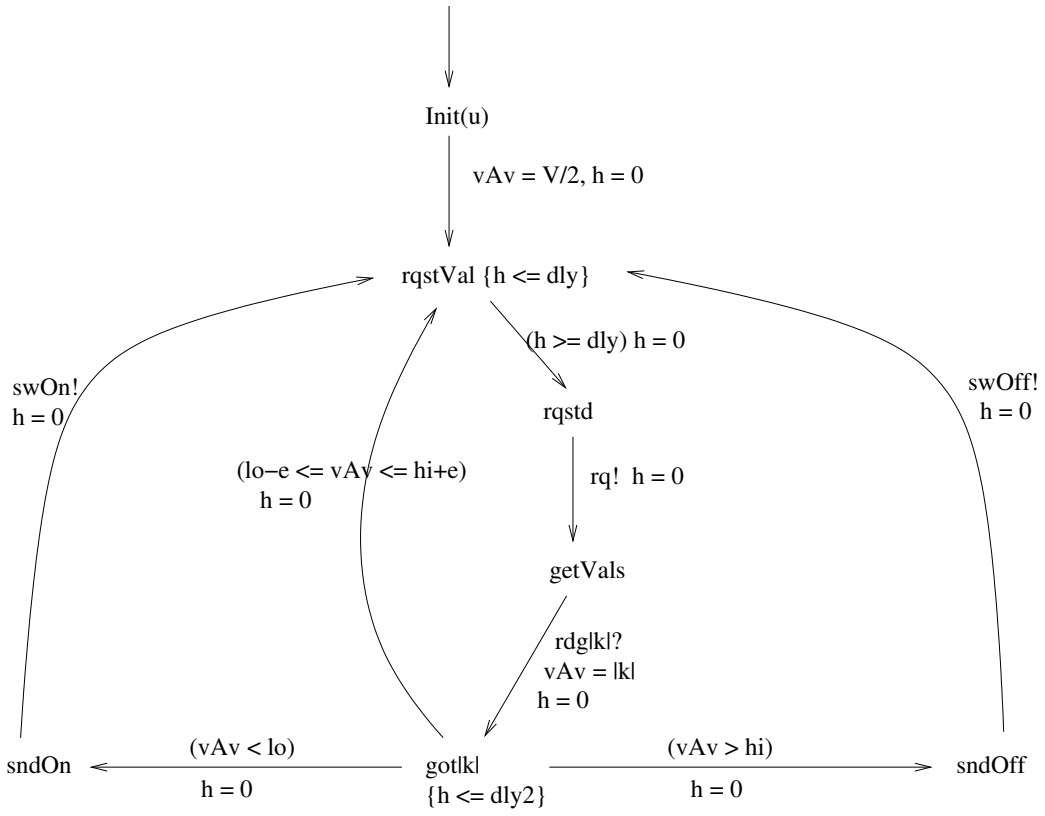


Figure 8.10: An abstract controller process

Similarly, $\mathbb{B}$ and $\mathbb{B}'$ have the same locations and transition edges, save that some guards in $\mathbb{B}$ are stronger than the corresponding guards in $\mathbb{B}'$, from which similarly one infers a simulation from $\mathbb{A} \parallel \mathbb{B} \parallel \mathbb{C}$ to $\mathbb{A} \parallel \mathbb{B}' \parallel \mathbb{C}$.

Thus, the premisses of theorem 7.5.2 are fulfilled and $\mathbb{A} \parallel \mathbb{B} \parallel \mathbb{C} \preceq \mathbb{A}' \parallel \mathbb{B}' \parallel \mathbb{C}$.

This implies that all traces of $\mathbb{A} \parallel \mathbb{B} \parallel \mathbb{C}$ are therefore possible traces of $\mathbb{A}' \parallel \mathbb{B}' \parallel \mathbb{C}$ and since $\forall\Box(\text{controller.vAv} < \xi)$ is (for suitable ξ) satisfied by $\mathbb{A}' \parallel \mathbb{B}' \parallel \mathbb{C}$, it is also by $\mathbb{A} \parallel \mathbb{B} \parallel \mathbb{C}$.

Likewise, $\forall\Box(\text{controller.getVals} \Rightarrow \text{controller.vAv} > \xi)$ (for suitable ξ) and $\forall\Box\neg$deadlock are satisfied by $\mathbb{A} \parallel \mathbb{B} \parallel \mathbb{C}$.

Now we know that all universally quantified properties of the abstraction are also properties of the original, we can test the properties on the abstraction $\mathbb{A}' \parallel \mathbb{B}' \parallel \mathbb{C}$. The results, as before presented as averages of 20 trials, are

as follows. Parameter `e` is set to 2 as before.

check:	$o(12)$	$o(14)$	$o(16)$	$u(24)$	$u(26)$	$u(28)$
mean user time:	30.8”	30.9”	15.7”	28.8”	30.52”	32.3”
standard dev :	0.1	0.5	0.1	0.2	0.1	0.4
mean max v mem:	44.791	44.791	42.552	44.624	44.790	44.792
standard dev :	0.003	0.003	0.002	0.058	0.002	0.002
mean max rss:	10.010	10.010	7.874	9.863	10.010	10.099
standard dev :	0.005	0.002	0.005	0.005	0.006	0.005

Now the time to check for deadlock-freeness is down from 41 minutes to 172.7 seconds (averaged over 20 trials with standard deviation 7.6). Memory usage was similar: max v mem = 48.7 Mb (SD = 0.003), rss = 14.1 Mb (SD = 0.002).

This abstract system is deadlock-free, of course, and also the o(12), o(14), u(26) and u(28) properties were verified. However in this abstraction, the properties $o(16)$, $u(24)$ fail when `e` set to 2. This is not in fact a very surprising result: it is telling us that the abstraction will not keep the variable controller.vAv within 2 of the set point when `e`=2: there is too much “slack” in the system. Notice the checks on $o(16)$, $u(24)$ took less time than the other checks in this test: the o(16) check was especially quick, presumably because UPPAAL `verifyta` found a counter-example very quickly.

We now have a situation where we can infer deadlock-freeness and also o(12), o(14), u(26) and u(28) for the original more “concrete” process (figures 8.7, 8.8) an order of magnitude faster than by checking it directly. However the failure of $o(16)$, $u(24)$ leaves us with no help with the original process. In view of the fact that these properties were verified in the original check we evidently have in the abstract model *spurious* counterexamples for them.

When `e` = 1 in the, all properties, o(12), o(14), o(16), u(24), u(26) and u(28) and deadlock-freeness, were verified. The $o(16)$ check of the abstract model took 21.9 seconds (averaged over 20 trials; SD = 0.2) and the $u(24)$ check took 21.5 seconds. By comparison, checks of these two properties of the concrete model took 260.1 seconds (SD = 4.3) and 259.2 seconds (SD = 8.7) respectively, over an order of magnitude longer.

This abstraction has, overall, yielded a significant speed-up of checks of $\forall\Box$-properties of the original model. A limitation is that it is possible for a properties to be falsified by the abstraction even when true in the concrete model: the abstraction can yield spurious counterexamples.

Nevertheless, the example illustrates how compositional reasoning can be used to speed up the checking of an embedded system model: decompose it and replace components with abstractions which the original components simulate; use compositional theorems and reasoning (such as assume-guarantee) to infer an overall simulation relationship and hence infer properties of the system model from the same properties of its abstraction.

8.3.2 Broadcasting

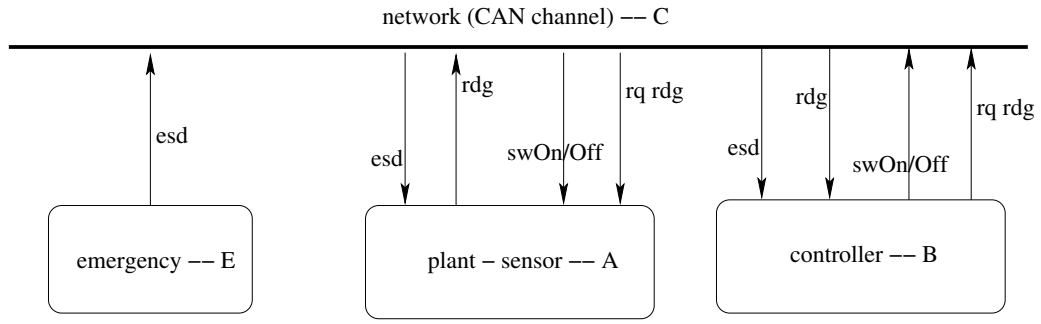


Figure 8.11: A system with broadcast emergency-shutdown

A slight extension of this scenario introduces broadcast – a sender sending to multiple receivers. In figure 8.11, The system $\mathbb{A} \parallel \mathbb{B} \parallel \mathbb{C}$ is extended with a process $\mathbb{E}$ which simulates an emergency, causing it to emit an urgent "emergency shut-down" synchronisation `esd` – see figure 8.12.

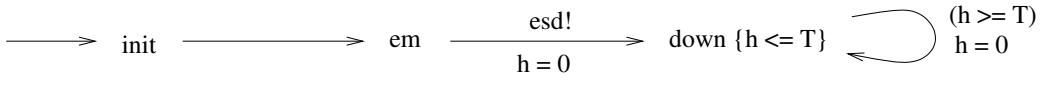


Figure 8.12: The emergency simulator

Processes $\mathbb{A}$ and $\mathbb{B}$ both receive this broadcast and are accordingly ex-

tended each with an extra location `down` and a transition from every other location with a broadcast synchronisation with the emergency shutdown message.

It can be checked that the broadcast emergency shut-down works as intended: $\forall\Box(\texttt{emsd.down} \Rightarrow \forall\Diamond(\texttt{controller.down} \wedge \texttt{plantSensor.down}))$ where `emsd` denotes an instance of the emergency simulator. Using the abstractions $\mathbb{A}'$, $\mathbb{B}'$ in lieu of $\mathbb{A}$, $\mathbb{B}$, this property was verified for the system $\mathbb{A}' \parallel \mathbb{B}' \parallel \mathbb{C} \parallel \mathbb{E}$ using the same hardware set as before in 35 minutes, 52.1 seconds. The $o(\xi)$, $u(\xi)$ and non-deadlock properties were still satisfied also.

Unfortunately, the assume-guarantee theorem is no longer useful, for in order to infer $\mathbb{A} \parallel \mathbb{B} \parallel \mathbb{C} \parallel \mathbb{E} \preceq \mathbb{A}' \parallel \mathbb{B}' \parallel \mathbb{C} \parallel \mathbb{E}$ one needs this already in order to satisfy premiss (b) of the theorem.

A limitation of the assume-guarantee theorem is that processes jointly participating in a synchronisation (such as $\mathbb{A}$, $\mathbb{B}$, $\mathbb{C}$ in the present instance) cannot be simulated separately, but must be simulated (or abstracted) together.

8.4 Working with bCANdle

So far, our examples have focussed on systems modelled by UPPAAL-type automata, communicating via a CAN channel. These are interesting in that they show how compositional reasoning can be applied to UPPAAL models of embedded systems communicating by CAN. However our starting point was the bCANdle specification framework.

The example of section 8.3, however, uses constructs which can all be defined in bCANdle.

A process very similar to the plant-sensor process of (figure 8.7) can be specified as a bCANdle process term `PlantSensor` by the equations below, in which g, h are clock variables, v an integer variable, on a boolean, and p an integer parameter.

$\texttt{PlantSensor} = [g = h = 0, v = V/2, on = 0 : 0, 0]$; `PSDutyCycle`

$\texttt{PSDutyCycle} = [\text{idle: } p/2, p]$
$\quad [> \texttt{ChgState}; \texttt{PSDutyCycle}$
$\quad [> \texttt{RecOffOn}; \texttt{PSDutyCycle}$
$\quad [> \texttt{Comm}; \texttt{PSDutyCycle}$

$\texttt{ChgState} = (on \rightarrow [v++, h = 0 : p/2, p] + !on \rightarrow [v--, h = 0 : p/2, p]$
$\quad +[h = 0 : p/2, p]$
$\texttt{RecOffOn} = ch?1.s; (s == 0 \rightarrow [on = 0, g = 0 : 0, 0]$
$\quad +s == 1 \rightarrow [on = 1, g = 0 : 0, 0])$
$\texttt{Comm} = \texttt{RecRq}; \texttt{SendVal}$
$\texttt{RecRq} = ch?2.x[> (\texttt{RecOffOn}; \texttt{RecRq})$
$\texttt{SendVal} = \sum_k (v == k \rightarrow ch!0.k)[> (\texttt{RecOffOn}; \texttt{SendVal})$

Here, we assume a data environment D_{PS} which supports integer variable v (whoe type is a subrange from 0 to some maximum V) and boolean variable on, basic operations to initialise these varaibles and the clocks, a predicate to test whether v has value k ($k = 0, ..., V-1$) and basic operations to increment and decrement v.

Symbol p is a time-delay parameter corresponding to the p in figure 8.7, and ch denotes the CAN channel. Also recall that a message of type 0 sends a variable reading, a message type 1 sends a command to switch on or switch off, and a message of type 2 requests a reading.

Similarly, to define a process like the controller of figure 8.8 would require a bCANdle equational specification of the form

$\texttt{Controller} = [g = h = 0, v_{av} = V/2 : 0, 0]$; $\texttt{ControllerDutyCycle}$

$\texttt{ControllerDutyCycle} =$
$\quad [\omega_{null} : dly, dly];\ ch!2.0;\ ch?0.v_{av};$
$\quad ((\gamma_{lo} \rightarrow ch!1.1) + (\gamma_{med} \rightarrow [\omega_{null} : 0, dly_2] + (\gamma_{hi} \rightarrow ch!1.0));$
$\quad \texttt{ControllerDutyCycle}$

As with the plant-sensor process we assume given a data environment D_C

with support for the integer variable v_{av}, a basic operation to initialise v_{av} and the clocks g, h, operation ω_{null} which does nothing but reset clock h on completion, and predicates

$\gamma_{lo} \equiv (v_{av} < lo \wedge g > inert) \vee v_{av} < lo - e,$
$\gamma_{hi} \equiv (v_{av} > hi \wedge g > inert) \vee v_{av} > lo + e,$
$\gamma_{med} \equiv ((v_{av} \geq lo \wedge v_{av} \leq hi) \vee (g \leq inert \wedge (v_{av} \geq lo - e \wedge v_{av} \leq hi + e)))$

to test the value of variable v_{av} in relation to the values of parameters lo, hi, e and the clock g in relation to the inertia parameter. Parameters $lo, hi, e, inert$ are as depicted in figure 8.8 and discussed in §8.3. The clock resets $h = 0$ appearing in 8.8 add nothing to the functionality of the model but do potentially aid verification by reducing the number of states needing to be checked.

A bCANdle specification (P, N, D) analogous to the example would be constructed as the parallel composite $P =$ `PlantSensor|Controller` with a network term N containing the channel k and data environment $D = D_{PS} \sqcup D_C$.

This example shows how, starting from a bCANdle specification, a parallel composite of timed automata can be constructed using the constructions employed in the proof of theorem 7.4.2: see § 5.4.2 and appendix B).

Developing `canGen`. To do this is always possible, but to do it by hand is laborious, of course. An interesting future project is to develop a software tool, an extension of `CANGen`, to parse a bCANdle specification and generate an equivalent composite of timed automata, an equivalent UPPAAL "system".

Parsing bCANdle process terms is straightforward, as the syntax is defined by a BNF grammar. As the examples above and in §5.4.2 show, the bCANdle data environment D accompanying a process term must be inferred at a suitable level of abstraction from from the variables and atomic operations occurring in the process terms. a CANGen-generated channel of m message types and n values The tool will then have to generate from the parse tree the clocked bCANdle terms (with associated data environment states) and transitions between them as illustrated in the example of §5.4.2.

The rules of this section will need to be recursively applied to generate the a timed automaton equivalent to the the original clocked process term and data environment. There will be opportunities for optimising the automaton, for instance by resetting clocks in order to trim the state space size. A system defined as a bCANdle system comprising a set of parallel processes not employing parallelism at a lower level (a restriction also imposed in the original bCANdle analysis) will thus give rise to a set of parallel timed automata communicating by CAN channels. An extended version of `CANGen` expand this into an UPPAAL "system".

A possible further development of this tool would be to integrate it with a model checker rather than using UPPAAL, and to incorporate counter-example-guided abstraction refinement (CEGAR). The latter is reported ([44], see below) to be a powerful model-checking technique in itself. Because of the essential role of "human" reasoning in obtaining compositional abstractions via simulations of components, such a tool will need to output timed automata (in some suitable) language and accept such timed automata as well as bCANdle systems as input.

8.5 Scope and Limits of the Method

The method described in this chapter will in theory extend to as many CAN channels, message types and payload values as one would wish, although the present version of CANGen generates only a single channel automaton. This has sufficed to prove the concept of the approach and an upgrade of CANGen to generate multiple channels is future work. Of course, the definition of a CAN frame allows 2^{11} message types and 2^{64} data values. One could in theory thus have for a single channel 2^{75} urgent synchronisations (input), broadcast sychronisations (output) and associated indexed automaton locations: theoretically possible, but well beyond what is practicable using current computing technology.

Alur [15] reports that the computational time-complexity of reachability analysis of the state space of a timed automaton with n locations and k clocks, in which every clock constraint constant is $\leq c$, is $n.2^{O(klog(kc))}$ (cf

§3.3.1), the region automaton has $n.k!.2^k.(2c+2)^k$ vertices and that the reachability problem is PSPACE-complete. On the other hand, the space- and time-complexity of expanding a CANGen source is linear both in the number of of message types and in the number of message values. Thus the complexity of the combined verification is essentially the complexity of the back-end model-check.

Figure 8.13: Simple sender and Receiver

The practical limit of the method is therefore at present determined by UPPAAL and the complexity of the systems it can check. As a simple test, checks were tried on a simple system in which a CANGen-generated channel of m message types and n values was fed every 90–100 time units by a simple sender process with a message of arbitrary type μ ($0 \leq \mu < m$) and value ν ($0 \leq \nu < n$), and a simple receiver process took delivery of it (figure 8.13). The UPPAAL property `sender.`$\mathtt{L}_{\mu,\nu}$ `--> receiver.`$\mathtt{R}_{\mu,\nu}$[3] was checked for various values of m, n, μ, ν.

With ten message types and 50 values ($m = 10,\ n = 50$) the property `sender.`$\mathtt{L}_{8,10}$ `--> receiver.`$\mathtt{R}_{8,10}$ was verified in around 75 seconds. With $m = 5,\ n = 120$ the property `sender.`$\mathtt{L}_{4,60}$ `--> receiver.`$\mathtt{R}_{4,60}$ was verified in around 150 seconds. However, doubling either the number of message types or the number values modelled overwhelmed the simple resources that have been used for the experiments described in this chapter. For example, the following caused the verification run to thrash -

- $m = 5,\ n = 240$: `sender.`$\mathtt{L}_{4,120}$ `--> receiver.`$\mathtt{R}_{4,120}$
- $m = 5,\ n = 180$: `sender.`$\mathtt{L}_{4,120}$ `--> receiver.`$\mathtt{R}_{4,120}$

[3] ie $\forall\Box(\texttt{sender.}\mathtt{L}_{\mu,\nu} \Rightarrow \forall\Diamond\texttt{receiver.}\mathtt{R}_{\mu,\nu})$

- $m = 10,\ n = 100$: `sender.`$\mathtt{L}_{8,90}$ `-->` `receiver.`$\mathtt{R}_{8,90}$
- $m = 10,\ n = 75$: `sender.`$\mathtt{L}_{8,60}$ `-->` `receiver.`$\mathtt{R}_{8,60}$

The method is clearly restricted to rather small numbers of message types and values, although the boundaries might be pushed a little by use of more powerful computing equipment (or at least, more RAM). Future work might include adaptation of the method to model checkers other than UPPAAL, or perhaps, even, a custom checker. However, it is unlikely we shall be able accommodate very large values of m, n.

On the other hand, do we need to? The method is intended to work with CAN-communicating systems at an abstract level and although an implemented system might well communicate numerical values from a large range of values, an abstraction of it might well (depending on the logic of the system) simulate it adequately using a much smaller range – small enough to be accommodated within the limits implied by these tests.

As discussed briefly in §1.3, methods other than model-checking with timed automata might well be appropriate for checking systems at a low level of implementation detail, such as true, as opposed to interleaving concurrency methods. The present work focuses on modelling timed automata, however, as this is widely used for checking properties at a high level of abstraction. The approach is suitable for working at such an abstract level with, for instance, communication protocols or distributed control systems in which the potential number of states is bounded and where one is interested in properties such as

- *safety* properties: for instance, a variable is maintained within a desired range;
- *liveness* and *bounded* liveness: for example, the system responds to an input from its environment or delivers output to its environment *within some time bound*;
- *fairness* and freedom from *starvation*: for instance, actions or processes that are required to be performed with minimum frequency are.

The approach of this thesis is *not* intended for analysing aspects of system performance such as "throughput", speed, bandwidth. It focusses pretty much on the ability of a system to respond and to meet real-time deadlines.

It was mentioned in §3.3.1 that forward reachability analysis allows a tool like UPPAAL to check many properties other than those which can be expressed directly in its specification language, by means of *test automata*. These allow one to formulate checks for safety, liveness and fairness properties as above – indeed anything that may be formulated in *timed* TCTL (§3.3.2). Also, as Bowman and Gomez [29] point out in their chapter 11, reachability analysis is versatile: many general liveness properties can be handled by nested reachability checks, although this is computationally expensive.

Of course, when one is making use of simulation relations in order to employ compositionality, we must confine ourselves to universal quantification over paths.

As it stands, the method used in the present work is a mixture of manual and automatic procedures. A simple prototype translator exists which, starting with a bCANdle system, generates equivalent CANGen source code. This tool (CANTranslator on the accompanying CD) is at a very early stage of development, defined earlier in the thesis as future work, and has not figured in the experiments of the present chapter. The CANGen source code, explained in §8.1, can on a small scale be easily written by hand working from timed automata or translating from bCANdle according to the operation semantic rules of bCANdle (as in the example at the end of §5.4.2).

The source code is expanded by CANGen into UPPAAL `.xta`-format code which can be checked by UPPAAL's `verifyta` tool. CANGen at present supports one CAN channel: this has sufficed for our experiments but a upgrade is envisaged for "production" use.

Thus far, a mixture of automatic and manual methods has be employed but it is envisaged that in the future this part of the method will be fully automated. Constructing component-wise abstractions via simulation relations is, on the other hand, a matter of human creativity. One needs to study the logic a particular compositional model in order to find suitable abstractions which simulate its components.

J Berendsen and F Vaangrager [27], exploring compositional abstraction in a more general setting than the present work, do suggest some approaches to this. They suggest -

- Weaken guards and location invariants of a component to obtain an "overapproximating" abstraction;
- After weakening guards and invariants, omit variables that are no longer tested;
- It may be possible to keep one component (or a small number of components) exact and replace the remainder by a single very coarse abstraction, preserving just the information about their interaction with the one "in the spotlight". B Wachter and B Westphal [112] develop this approach, which they call the "Spotlight Principle";
- It may be possible to abstract a component automaton by a parallel combination of automata; although Berendsen and Vaandrager found proving the correctness of an instance of this hard.

The examples discussed earlier in this chapter (§§8.2, 8.4) employed reasoning of the type covered by the first two items.

This element of the approach will inevitably require some manual input but possible future work will explorer automated methods such as counterexample-guided abstraction refinement.

8.6 Summary

An embedded system composed of parallel components communicating by CAN can be modelled in bCANdle. The several components of this model can be translated into timed automata as explained in §5.4.2, using the rules set out there. The translation is in fact into the formalism comprising the input to `CANGen`. As just discussed, this procedure can (and eventually will) be implemented in a software tool. The result is a composition of parallel timed automata including automata which model the CAN channel(s). After

expansion with `CANGen` the result is an UPPAAL "system", whose properties expressible in ACTL can be checked.

In the case of models which are unfeasibly large for this, there is the opportunity for replacing components of the model with (usually more abstract) components which *simulate* them. The framework for doing this is provided by the theorems of chapter 7 and examples explored in the preceding sections of this chapter. Finding *useful* simulations will usually be a matter of human ingenuity although, as seen in §8.3, *abstracting* data and operations of the model is generally a promising strategy. *Counterexample-guided abstraction refinement* is a promising "automatic" technique for refining such simulations when a property is found to fail: this possible refinement of our work is discussed briefly in the next chapter.

Compositional models can be developed directly as timed automata, as well as bCANdle models; and the framework devloped here could be ported to other kinds of communicating systems, as long as the communication mechanism is modellable as a timed automaton.

Chapter 9

Conclusions and Further Work

9.1 Model Checking Broadcasting Embedded Systems

This thesis has examined the used of compositional methods for checking models of broadcasting embedded systems comprising processes communicating via CAN network channels. In particular, it has examined D Kendall's bCANdle formalism (chapter 4), and shown (chapter 5) how it can be cast into an equivalent form in which all the system components are (parallel) timed automata modellable in UPPAAL.

Thus, a system expressible in bCANdle can be modelled as a parallel composition of UPPAAL timed automata. This representation was further cast into an equivalent form (chapter 6) in which variables were not shared between processes. This allowed some compositionality results (chapter 7) to be applied.

The compositionality results allow one to infer a *simulation* of a given composite model $\mathbb{A} \parallel \mathbb{B} \parallel \mathbb{C} \parallel ...$ by some more abstract model $\mathbb{A}' \parallel \mathbb{B}' \parallel \mathbb{C}' \parallel ...$ for which checking of some property φ might be more tractable. The simulation relation implies a *trace inclusion*

$$\mathbb{A} \parallel \mathbb{B} \parallel \mathbb{C} \parallel ... \preceq \mathbb{A}' \parallel \mathbb{B}' \parallel \mathbb{C}' \parallel ...$$

and so the properties φ need to be universally quantified over times and paths (of the form $\forall\Box\psi$ in CTL, for instance). Since we are dealing with timed systems, this is not unduly restrictive. For instance, *absence* of deadlock can be cast into an assertion that in every run, every state is *non*-deadlocking.

In particular, *assume-guarantee* theorems such as that of §7.4.2 allow one to infer such a trace-inclusion relation between composite systems by abstracting "component-wise". An example of this was described in section 8.3. The premiss (b) of the assume-guarantee theorem 7.5.2 is a tough requirement, however, and some creativity is required to devise models that satisfy it. Useful further work includes a possible strengthening of the theorem by some sort of relaxation of this premiss.

In the example just cited, the method was applied to a composite model built in terms of timed automata; it can *a fortiori* be applied to bCANdle models, as discussed in section 8.4. It is hoped that the results presented here have wider applicability and interest than in systems employing CAN networks; nevertheless these are an important class of embedded systems and the present compositional formulation is a useful adjunct to bCANdle.

The models explored in chapter 8 were reasoned about by means of abstractions which had the same timed-automaton structure as their more refined, detailed counterparts, but weaker guards on some transitions. This turned out to be a powerful way of obtaining the simulation relations useful in compositional reasoning. One can verify a property of a (component of a) system by exhibiting the system as a "more determined" version of some abstract non-deterministic model (with the same locations and weaker transition guards) which nevertheless has the required behaviour.

9.2 Model Checking More Generally

The compositional reasoning illustrated by the examples in chapter 8 is a useful adjunct to any method that uses timed automata to model concurrent real-time systems. Models of practical systems become intractable when scaled up to "real-world" size because of the "state explosion" problem which besets them. This was found, for instance, in early attempts by the present

author to model a simple robotic manufacturing cell with a conveyor that could stop and start. Compositional reasoning provides a way of dealing with state explosion by allowing one to "divide and conquer" by using abstraction.

The models discussed in chapters 5, 6, 8 were models of systems employing CAN, but were built directly using UPPAL-type timed automata. In fact nothing in the present work implies that it *has* to apply *only* to these. In fact the compositional reasoning illustrated in chapter 8 could apply to any system modellable in UPPAAL, or indeed, with "plain" timed automata, in a way which is compatible with the compositionality theorems. One could, for instance, model distributed systems using a network technology other than CAN, provided only that the network behaviour can be modelled as a timed process.

9.3 Further Work

A number of papers, while not bearing directly on the work of this thesis, do suggest avenues of future investigation.

Alexander Rabinovich [106] defines a generalised parallel product of which many parallel compositions are special cases, and obtains positive results for a compositional method based on it, at least for "basic propositional modal logic". G Frehse [57] has worked in a similar way to the present work, using hybrid automata communicating using discrete events and without shared variables; this work does not use UPPAAL or deal with CAN broadcasting but offers some interesting future directions. Rodolfo Gmez [61] develops a representation of urgent actions using *timed automata with deadlines* which avoids "timelocks" and provides a compositional translation of these into UPPAAL. A Gupta, K L McMillan and Z Fu [64] use a boolean satisfiability solver in assume-guarantee reasoning which they say outperforms BDD-based model checking. Junyan Qian, Lingzhong Zhao, Guoyong Cai and Tianlong Gu [105] present *formula-dependent* abstraction for CTL which does not preserve ACTL formulae as simulation does, so might be too coarse, but does, they say, reduce the size of the Kripke structure; and they provide an accompanying refinement method. Timothy Bourke and Arcot Sowmya

[28] are interested in the fact that a deterministic τ-free timed automaton can be transformed into a form where reachability analysis can decide trace-inclusion in another automaton. They remark that manual transformation is tedious and error-prone and provide a tool for automatically transforming UPPAAL automata.

9.3.1 Counterexample-guided Abstraction Refinement

Section 3.5 described a method developed by E Clarke, O Grumberg, S Jha, Y Lu and H Veith [44] for making and then refining abstractions of a model. If a universally quantified property holds in the abstraction then it holds also in the original model as execution traces of the original model are included among traces of the abstraction. If the property does *not* hold in the abstraction, the method produces a counterexample which *either* pulls back to a counterexample in the original model *or* is *spurious*, in which case the method suggests a *refinement* of the abstraction which eliminates at least one element of spuriousness. The CEGAR approach is surveyed by Orna Grumberg in [62].

This method was developed for automatic use in model-checking tools for untimed systems specified by ACTL* formulae – indeed, the authors describe how their algorithms are implemented using data structures such as ordered binary decision diagrams (OBDD). However, their algorithms and their reasoning are couched entirely in terms of states of a Kripke structure, sets of states and transitions between states.

The timed transition systems of *our* models are in fact Kripke structures whose states are vectors of variable values. Indeed, in §§7.2.1, 7.2.2 it was shown how the structures of Kaynar, Lynch *et al.* [77, 78] subsume the timed transition systems of our models. They are easily endowed with the Kripke structure: they have a set of states (which are vectors of variable values), an initial state subset (usually a singleton), a transition relation, and the function L can be set in the same fashion as in Clarke *et al.* [44], mapping a state to the set of "atoms"[1] of the system true there.

[1] atomic subformulae of the guards of the system, and their negations; cf. §3.5.1

A timed automaton (in *our* sense) can be seen as a "concurrent program" in the sense of Clarke *et al.*: see §3.5.1, page 64. In the notation used there, each variable v_i is associated with an initialiser I_i derived from the automaton and undergoes an update A_i^k when a guard C_i^k is satisfied. The A_i^k and C_i^k are again derived from the guards and updates of the automaton. The timed transition system of one of our models is thus, in effect, a Kripke structure of the type constructed by Clarke *et al.*

The reasoning set out in §3.5.1 leading to the construction of an *existential abstraction* of the timed transition system based on an equivalence relation between states extends in a straightforward way to the timed transition systems of our models. Thus we can envisage an existential abstraction $\mathbb{M}'$ of a model $\mathbb{M}$ of our type, such that $\mathbb{M} \preceq \mathbb{M}'$, $\mathbb{M}$ is *simulated by* $\mathbb{M}'$.

Spurious counterexamples arise in these abstract models in the fashion described in §3.5.3 and can in principle be detected by an algorithm like the `SplitPATH` described in that section. Abstractions containing spurious counterexamples can be refined using the procedures of Clarke *et al.* described in §3.5.4.

This approach was developed by Clarke *et al.* for use in an automatic (software) model-checking tool working with efficient representations of states (using ordered binary decision diagrams) of the timed transition system of the model. To apply it to our framework would similarly require detailed examination of timed states: not suitable for working "by hand" but worth incorporating in a software model-checking tool for our framework.

Chao Wang, Hyondeuk Kim, Aarti Gupta [113] present a hybrid method combining CEGAR with variable-hiding methods to obtain abstraction methods which they claim perform better than previously existing ones.

Our approach has been to build models in a "macro" language which is preprocessed by our `CANGen` into input to UPPAAL's `verifyta` tool. It does indeed seem a worthwhile future enterprise to streamline this process into a single tool which will verify the model directly. Counterexample-guided abstraction refinement will form a useful adjunct to this.

A number of more recent papers suggest a variety of novel approaches to generating and refining abstractions and also assumptions for use in auto-

mated assume guarantee reasoning, including use of machine learning, theorem proving and syntactic transfomrations, and SAT solvers: J M Cobleigh, G S Avrunin and L A Clarke [48]; R Ben Salah, M Bozga and O Maler [107, 21]; N Sharygina, S Tonetta and A Tsitovich [108]; V D'Silva, S Sonalkar and S Ramesh [52]; S Kundu, S Lerner and R Gupta [86]; B Finkbeiner, H Peter, and S Schewe [55]; H Yao and H Zheng [116]; C Pasareanu and D Giannakopoulou [103]; Y Meller, O Grumberg and S Shoham [96]; E Clarke, M Talupur and H Veith [47]; B Finkbeiner, S Schewe and M Brill [56]; S Bensalem, M Bozga, J Sifakis and Thanh-Hung Nguyen [24, 25]; W Nam, P Madhusudan and R Alur [100, 18].

9.3.2 Stopwatch and Hybrid Automata

We have spent the most time in the world of *timed* automata; but early invesigations of compositional model checking problems looked at *linear hybrid automata* (Cassez, Larsson [43], Alur *et al.* [3]). These are "classic" timed automata extended with variables which vary at a constant rate in time, may be updated in a mathematically *linear* fashion on discrete actions, and linear combinations of which may feature in guards. Special cases include *stopwatch automata*, timed automata with clocks that can stop and start. Hybrid automata have not figured explicitly in the later work here as they are subsumed by UPPAAL, supporting as it does a rich set of data types. Interestingly, stopwatches are also straightforward to model in UPPAAL, but models that incorporate them are notoriously difficult to check. Compositional techniques may provide a way into this.

The linear hybrid systems of Thao Dang ([49]) are an altogether more ambitious construction, incorporating as they do continuous variables whose values evolve according to linear differential equations. Interestingly she has worked with predicate abstraction, including counterexample-guided predicate abstraction, in collaboration with Alur and F Ivancic [5, 6]. It would be interesting to see if the framework developed in this thesis extends to them.

9.3.3 Hybrid I/O Automata

These are a development by N Lynch, R Segala and F Vaandrager [94] of their Timed I/O Automata, discussed in §§3.4.3, 7.2.2. The main innovation is that their structures not only distinguish discrete *internal* and *external* discrete actions, but also *internal* and *external* (continuous) variables. In their hybrid *I/O* automaton the external variables are further partitioned into *input* and *output* variables – essentially, variables which might be passed into a process and which might return data from a process, respectively (internal variables are local to a process). UPPAAL supports something like this, as it provides reference parameters to a process; but we have gone to quite a bit of trouble to avoid shared variables between processes in our framework, so it is a moot point whether this will usefully extend it.

One possibility is that the compositionality theorems may be sharpened to versions that do apply to systems with shared variables.

9.3.4 Separation Logic and Rely-Guarantee Reasoning

Rely-guarantee reasoning is an approach to verifying properties of concurrent programs (see e.g. V Vafeiadis [110], Vafeiadis and Parkinson [111], Xu *et al.* [115]) that is reminiscent of assume-guarantee reasoning.

Vafeiadis and Parkinson, for instance, deal with formulae of the form PCQ in which P, Q are predicates on the program state and C a "command", some sort of processing. The formula's intended meaning is then, "if P was true of the program state immediately before C is issued, Q is true of the state immediately after". Formulae of this type can be compared with the assume-guarantee formulae $\langle\varphi\rangle\mathcal{M}\langle\psi\rangle$ briefly discussed in §3.4.1.

Vafeiadis and Parkinson use the *separation logic* of Peter O'Hearn and his collaborators [101, 102] which extends the syntax and semantics of P, Q by adding to classical boolean logic a "separating conjunction" $P * Q$, true in a state if the state separates into two disjoint states, one satisfying P and one satisfying Q. There is also a "separating implication" $P -\!\!* Q$ true at a local state s if any state disjoint from s and satisfying P results in a conjoined state satisfying Q.

O'Hearn *et al.* develop axioms and rules of inference for a separation logic which allows them to "prove" a concurrent program by deriving formulae of the form $\{P\}C\{Q\}$ where C is a program command with pre-condition(s) P and post-condition(s) Q. Vafeiadis and Parkinson extend this work in [111] by carefully distinguishing the *local state* from the *shared state* of a process and developing from this a notion of *interference* in a command C by its *environment*. They define a formula

$$\vdash \; C \; \texttt{sat}(p, R, G, q)$$

in which p is a preconditon of command C, q a post-condition, and R, G are sets of actions. The intended meaning is that if (1) the execution of C from an initial state satisfying p and under interference $\subseteq R$ (R forms a *rely condition*) does not fault, and (2) this execution causes interference $\subseteq G$ and, and (3) it terminates, then the final state satisfies q. They develop a system of proof rules by which a set of concurrent programs or processes may be proved to satisfy such formulae, formalising the reliance of each command in each process on a bound R on interference it suffers, and asserting a guaranteed bound G on the interference it causes.

This work was motivated by the problem of proving correct a system of concurrent processes sharing memory. The work of this thesis, on the other hand, has addressed its problems by recasting them into a form in which variables are *not* shared. Future work could to investigate possible extensions to the theorems of chapter 7 to models with shared variables. If this turns out to be useful, some analogue of separation logic may play a useful role. *Local* rely-guarantee reasoning (X Feng, [53] may also prove an interesting extension.

9.3.5 Practical Work

The bulk of the present work has been a theoretical exploration of compositional techniques for verifying properties of CAN-broadcasting embedded systems. The main verification tool was UPPAAL, but a simple "front-end" tool, `CANGen`, was developed to ease the task of defining models without

shared variables. `CANGen` is rather crude – it needs upgrading if it is to support more than one CAN channel, for instance, and its input "macro language" could be developed. It is used in conjunction with UPPAAL's `verifyta` program.

A promising idea is to make an integrated tool which generates the model from the `CANGen` input language *or* from a bCANdle specification, and does the verification.

An interesting adjunct to such a tool would be support for counterexample-guided abstraction refinement – see the discussion in §9.3.1. This would allow us to evaluate the usefulness of this technique in the present framework.

9.3.6 Heuristics

As already mentioned, it transpired in course of exploring assume-guarantee reasoning that a good approach was to obtain a model as a "concrete case" with the same automaton structure and stronger guards, of a more abstract non-deterministic model with desirable properties. What is required are more case studies to uncover more heuristics of this kind.

9.3.7 Assume-Guarantee

As already mentioned, it would be useful to know whether theorem 7.5.2 can be strengthened by admitting in place of its premiss (b) some weaker premiss. At the moment, it seems that when multiple components participate in a broadcasting synchronisation, they *have* to be consider together, indivisibly, in any compositional reasoning.

References

[1] L Aceto, P Bouyer, A Burgueno, and K G Larsen. The power of reachability testing for timed automata. *Theoretical Computer Science*, 300(1-3):411–475, 2003.

[2] R Alur, C Courcoubetis, N Halbwachs, D Dill, and H Wong-Toi. Minimization of timed transition systems. In *Proceedings of the Third Conference on Concurrency Theory*, number 630 in Lecture Notes in Computer Science, pages 340–354. Springer, 1992.

[3] R Alur, C Courcoubetis, N Halbwachs, T A Henzinger, P-H Ho, X Nicollin, A Olivero, J Sifakis, and S Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.

[4] R Alur, C Courcoubetis, and T A Henzinger. Computing accumulated delays in real time systems. *Formal Methods in System Design*, 11:137–156, 1997.

[5] R Alur, Th Dang, and F Ivancic. Reachability analysis of hybrid systems via predicate abstraction. In *Fifth International Workshop on Hybrid Systems: Computation and Control*, 2002.

[6] R Alur, Th Dang, and F Ivancic. *Counterexample-Guided Predicate Abstraction of Hybrid Systems*, pages 208–223. Lecture Notes in Computer Science 2619. Springer, 2003.

[7] R Alur and D Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[8] R Alur and D Dill. *Automata-theoretic Verification of Real-Time Systems*, pages 55–82. Trends in Software Series. Wiley, 1996.

[9] R Alur, K Etessami, S La Torre, and D Peled. Parametric temporal logic for "model measuring". *ACM Transactions on Computational Logic*, 2:388–407, 2001.

[10] R Alur and T A Henzinger. Logics and models of real time: A survey. In *Proceedings of the Real-Time: Theory in Practice, REX Workshop*, pages 74–106. Springer, 1992.

[11] R Alur and T A Henzinger. A really temporal logic. *Journal of the ACM*, 41:181–204, 1994.

[12] R Alur and T A Henzinger. Reactive modules. In *Proc 11th IEEE Symposium on Logic in Computer Science*, pages 207–218. IEEE Computer Society Press, 1996.

[13] R Alur, T A Henzinger, and O Kupferman. Alternating time temporal logic. *Journal of the ACM*, 49:672–713, 2002.

[14] R Alur and D Peled. Undecidability of partial order logics. *Information Processing Letters*, 69, 1999.

[15] Rajeev Alur. Timed automata. In *11th International Conference on Computer-Aided Verification*, number 1633 in Lecture Notes in Computer Science, pages 8–22. Springer, 1999.

[16] Rajeev Alur, Costas Courcoubetis, and David Dill. Model checking for real time systems. In *Proceedings of the 5th Annual Symposium on Logic in Computer Science*. IEEE Press, 1990.

[17] Rajeev Alur, Costas Courcoubetis, and David Dill. Model checking in dense real time. *Information and Computation*, 104(1):2–34, 1993.

[18] Rajeev Alur, P. Madhusudan, and Wonhong Nam. Symbolic compositional verification by learning assumptions. In *In CAV*, Lecture Notes in Computer Science, pages 548–562. Springer, 2005.

[19] G Behrmann, A David, and K G Larsen. *A Tutorial on Uppaal*, pages 200–236. Lecture Notes in Computer Science 3185. Springer, 2004.

[20] G Behrmann, T Hune, and F Vaandrager. Distributing timed model checking – how the search order matters. In *Proceedings of the 12th International Conference on Computer Aided Verification*, number 1855 in Lecture Notes in Computer Science, pages 216 – 231. Springer, 2000.

[21] Ramzi Ben Salah, Marius Dorel Bozga, and Oded Maler. Compositional timing analysis. In *EMSOFT '09: Proceedings of the seventh ACM international conference on Embedded software*, pages 39–48, New York, NY, USA, 2009. ACM.

[22] J Bengtsson, B Jonsson, J Lilius, and W Yi. Partial order reductions for timed systems. In *Proceedings of the 9th International Conference on Concurrency Theory*, number 1466 in Lecture Notes in Computer Science, pages 485 – 500. Springer, 1998.

[23] J Bengtsson and Wang Yi. *Timed Automata: Semantics, Algorithms and Tools*, pages 87–124. Lecture Notes in Computer Science 3098. Springer, 2004.

[24] Saddek Bensalem, Marius Bozga, Thanh-Hung Nguyen, and Joseph Sifakis. D-finder: A tool for compositional deadlock detection and verification. In *CAV*, pages 614–619, 2009.

[25] Saddek Bensalem, Marius Bozga, Thanh-Hung Nguyen, and Joseph Sifakis. Compositional verification for component-based systems and application. *IET Software, Special Issue on Automated Compositional Verification: Techniques, Applications and Empirical Studies*, 4(3):181 – 193, June 2010.

[26] B Bérard, M Bidoit, A Finkel, F Laroussinie, A Petit, L Petrucci, Ph Schnoebelen, and P McKenzie. *Systems and Software Verification*. Springer, 2001.

[27] Jasper Berendsen and Frits Vaandrager. Compositional abstraction in real-time model checking. In *FORMATS '08: Proceedings of the 6th international conference on Formal Modeling and Analysis of Timed Systems*, pages 233–249, Berlin, Heidelberg, 2008. Springer-Verlag.

[28] Timothy Bourke and Arcot Sowmya. Automatically transforming and relating Uppaal models of embedded systems. In *Proceedings of the 8th International Conference on Embedded Software*, pages 59–68, Atlanta, Georgia USA, Oct 2008. ACM.

[29] Howard Bowman and Rodolfo Gomez. *Concurrency Theory: Calculi an Automata for Modelling Untimed and Timed Concurrent Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[30] J Bradfield and C Stirling. *Modal Logics and μ-Calculi: an Introduction*, pages 293–330. Elsevier, 2001.

[31] S Bradley, W Henderson, D Kendall, and A Robson. An application oriented real-time algebra. *Software Engineering Journal*, 9:201–212, 1994.

[32] S Bradley, W Henderson, D Kendall, and A Robson. A formally based hard real-time kernel. *Microprocessors and Microsystems*, 18(9):513–521, November 1994.

[33] S Bradley, W Henderson, D Kendall, and A Robson. Practical formal development of real-time systems. In *11th IEEE Workshop on Real-Time Operating Systems and Software, RTOSS '94, Seattle*, pages 44–48, May 1994.

[34] S Bradley, W Henderson, D Kendall, and A Robson. Integrating aorta with model-based data specification languages. In E Astesiano, editor, *Fundamental Approaches to Software Engineering*, volume 1382 of *Lecture Notes in Computer Science*, pages 54–71. Springer, April 1998.

[35] S Bradley, W Henderson, D Kendall, A Robson, and S Hawkes. A formal design and implementation method for real-time embedded sys-

tems. In P Milligan and K Kuchinski, editors, *22nd EUROMICRO Conference (EUROMICRO 96), Prague*, pages 77–84. IEEE, September 1996.

[36] S Bradley, W D Henderson, D Kendall, and A P Robson. Designing and implementing correct real-time systems. In H Langmaack, W-P de Roever, and J Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems FTRTFT '94, Lubeck, Lecture Notes in Computer Science 863*, pages 228–246. Springer-Verlag, September 1994.

[37] S Bradley, D Kendall, W D Henderson, and A P Robson. Validation, verification and implementation of timed protocols using AORTA. In Piotr Dembinski and Marek Sredniawa, editors, *Protocol Specification, Testing and Verification XV (PSTV '95), Warsaw*, pages 193–208. IFIP, North Holland, June 1995.

[38] S P Bradley, W D Henderson, and D Kendall. Using timed automata for response time analysis of distributed real-time systems. In A H Frigeri, W A Halang, and S H Son, editors, *24th IFAC/IFIP Workshop on Real-Time Programming (WRTP 99)*, pages 143–148, May 1999.

[39] R B Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[40] R B Bryant. On the complexity of vlsi implementations and graph representations of boolean functions with applications to integer manipulation. *IEEE Transactions on Computers*, 40(2):205–213, 1991.

[41] R B Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[42] F Cassez and F Laroussinie. Model-checking for hybrid systems by quotienting and constraints solving. In *Proceedings of the 12th International Conference on Computer Aided Verification*, number 1855 in Lecture Notes in Computer Science, pages 373–388. Springer, 2000.

[43] F Cassez and K Larsen. The impressive power of stopwatches. In *Proceedings of the 11th International Conference on Concurrency Theory*, number 1877 in Lecture Notes in Computer Science, pages 138–152. Springer, 1999.

[44] E Clarke, O Grumberg, S Jha, Y Lu, and H Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.

[45] E M Clarke, O Grumberg, and D Peled. *Model Checking*. MIT Press, 1999.

[46] E M Clarke and B-H Schlingloff. *Model Checking*, pages 1635–1790. Elsevier, 2000.

[47] Edmund Clarke, Murali Talupur, and Helmut Veith. Proving ptolemy right: the environment abstraction framework for model checking concurrent systems. In *TACAS'08/ETAPS'08: Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, pages 33–47, Berlin, Heidelberg, 2008. Springer-Verlag.

[48] Jamieson M. Cobleigh, George S. Avrunin, and Lori A. Clarke. Breaking up is hard to do: An evaluation of automated assume-guarantee reasoning. *ACM Transactions on Software Engineering and Methodology*, 17-2, 2008.

[49] Thao Dang. *Verication and Synthesis of Hybrid Systems*. PhD thesis, VERIMAG laboratory, Grenoble, 2000.

[50] A David. Uppaal2k: Small tutorial, 1998.

[51] David Dill. Timing assumptions and verification of finite state concurrent systems. In J Sifakis, editor, *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*. Springer, 1989.

[52] Vijay D'Silva, Sampada Sonalkar, and S. Ramesh. Existential abstractions for distributed reactive systems via syntactic transformations. In *EMSOFT '07: Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 240–248, New York, NY, USA, 2007. ACM.

[53] Xinyu Feng. Local rely-guarantee reasoning. *SIGPLAN Not.*, 44(1):315–327, 2009.

[54] Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Schedulability analysis using two clocks. In Hubert Garavel and John Hatcliff, editors, *Proc. of TACAS'03*, number 2619 in Lecture Notes in Computer Science, pages 224–239. Springer–Verlag, April 2003.

[55] Bernd Finkbeiner, Hans-Jörg Peter, and Sven Schewe. Synthesizing certificates in networks of timed automata. In *RTSS '08: Proceedings of the 2008 Real-Time Systems Symposium*, pages 183–194, Washington, DC, USA, 2008. IEEE Computer Society.

[56] Bernd Finkbeiner, Sven Schewe, and Matthias Brill. Automatic synthesis of assumptions for compositional model checking. In *FORTE*, pages 143–158, 2006.

[57] G Frehse. Compositional verification of hybrid systems with discrete interaction using simulation relations. In *2004 IEEE International Symposium on Computer Aided Control Systems Design*, pages 59 – 64, 2004.

[58] S Garavel, F Lang, and R Mateescu. An overview of cadp 2001. Technical Report RT254, INRIA, 2001.

[59] S Garavel, F Lang, R Mateescu, and W Serwe. Cadp 2006: A toolbox for the construction and analysis of distributed processes. In *Proceedings of the 19th International Conference on Computer Aided Verification CAV 2007*, 2007.

[60] R Bosch GmbH. CAN specification version 2.0. Technical report, Bosch, September 1991.

[61] Rodolfo Gómez. A compositional translation of timed automata with deadlines to uppaal timed automata. In *FORMATS '09: Proceedings of the 7th International Conference on Formal Modeling and Analysis of Timed Systems*, pages 179–194, Berlin, Heidelberg, 2009. Springer-Verlag.

[62] Orna Grumberg. Abstraction and refinement in model checking. In *Formal Methods for Components and Objects, 4th International Symposium*, pages 219–242, 2005.

[63] Orna Grumberg and David E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16, 1994.

[64] Anubhav Gupta, K. L. Mcmillan, and Zhaohui Fu. Automated assumption generation for compositional verification. *Form. Methods Syst. Des.*, 32(3):285–301, 2008.

[65] W Henderson, D Kendall, and A Robson. Improving the accuracy of scheduling analysis applied to distributed systems. *Real-Time Systems*, 20:5–25, 2001.

[66] W Henderson, D Kendall, A Robson, and S Bradley. Xrma: An holistic approach to performance prediction of distributed real-time can systems, 1998.

[67] T A Henzinger, P W Kopke, A Puri, and P Varaiya. What's decidable about hybrid automata. *Journal of Computer and System Sciences*, 57:94–124, 1995/1998.

[68] T A Henzinger, Z Manna, and A Pnueli. What good are digital clocks? In *Proceedings of the 19th International Colloquium on Automata, Languages, and Programming (ICALP)*, number 623 in Lecture Notes in Computer Science, pages 545–558. Springer-Verlag, 1992.

[69] T A Henzinger, X Nicollin, J Sifakis, and S Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111:193–244, 1994.

[70] T A Henzinger, S Qadeer, and S K Rajamani. You assume, we guarantee: Methodology and case studies. In *Proc 10th International Conference on Computer-aided Verfiication*, number 1427 in Lecture Notes in Computer Science, pages 440 – 451. Springer, 1998.

[71] G Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[72] G Holzmann. The model checker spin. *IEEE Trans Software Eng*, 23(5), May 1997.

[73] J E Hopcroft and J D Ullman. *Automata Theory, Languages and Computation*. Addison Wesley, 1979.

[74] ISO. ISO/DIS 11898: Road vehicles - interchange of digital information - Controller Area Network (CAN) for high speed communication. Technical report, ISO, 1992.

[75] C B Jones. *Specification and design of (parallel) programs*, volume Proceedings of IFIP '83, pages 321–332. North Holland, 1983.

[76] Bernhard Josko. Modular specification and verification of reactive systems. Technical report, University of Oldenburg, 1993.

[77] D Kaynar, N Lynch, R Segala, and F Vaandrager. The theory of timed i/o automata. *Technical Report MIT-LCS-TR-917, MIT Laboratory for Computer Science*, 2003.

[78] D Kaynar, N Lynch, F Vaandrager, and R Segala. Decomposing verification of timed i/o automata. In *Formal Techniques, Modelling and Analysis of Timed and Fault Tolerant Systems: Joint International Conferences on Formal Modeling and Analysis of Timed Systems, FORMATS 2004, and Format Techniques in Real-Time and*

Fault-Tolerant Systems, FTRTFT 2004, number 3253 in Lecture Notes in Computer Science, pages 84–101. Springer-Verlag, 2004.

[79] D Kendall. *Formal Modelling and Analysis of Broadcasting Embedded Control Systems*. PhD thesis, Newcastle University School of Computing Science, 2001.

[80] D Kendall, S Bradley, W Henderson D, and A Robson. A formal basis for tool-supported simulation and verification of real-time can systems. In *Proceedings of 4th International CAN Conference (iCC'97)*, pages 719–727, 1997.

[81] D Kendall, S Bradley, W Henderson D, and A Robson. bcandle: Formal modelling and analysis of can control systems. In *Proceedings of 4th IEEE Real Time Technology and Applications Symposium (RTAS'98)*, pages 171–177. IEEE Press, 1998.

[82] D Kendall, S Bradley, W Henderson D, and A Robson. Candle: A high level language and development environment for high integrity can control systems. In *Proceedings of 4th IEE Workshop on Discrete Event Systems*, pages 58–63, 1998.

[83] D Kendall, W Henderson D, and A Robson. Modelling and analysis of embedded control systems, 1998.

[84] D Kendall, W Henderson D, and A Robson. Space efficient reachability analysis for a value passing, timed process algebra, 1999.

[85] D Kendall, W Henderson D, and A Robson. Using sharing trees in the automated analysis of real-time systems with data. In *IEE Colloquium on Applicable Modelling, Verification and Analysis Techniques for Real-Time Systems (Ref. No. 1999/006)*, pages 6/1–4, 1999.

[86] Sudipta Kundu, Sorin Lerner, and Rajesh Gupta. Automated refinement checking of concurrent systems. In *ICCAD '07: Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design*, pages 318–325, Piscataway, NJ, USA, 2007. IEEE Press.

[87] F Laroussinie and K G Larsen. Cmc: A tool for compositional model checking of real-time systems. *Proc IFIP Joint Conf Formal Description Techniques and Protocol Specification, Testing and Verification*, pages 439–456, 1998.

[88] K G Larsen, F Larsson, P Pettersson, and W Yi. Efficient verification of real-time systems: Compact data structure and state space reduction. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, pages 14–24, 1998.

[89] K G Larsen, P Pettersson, and W Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1:134–152, 1998.

[90] K G Larsen, C Weise, W Yi, and J Pearson. Clock difference diagrams. *Nordic Journal of Computing*, 1999.

[91] Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Compact data structures and state-space reduction for model-checking real-time systems. *Real-Time Syst.*, 25(2-3):255–275, 2003.

[92] G Leduc, A Jeffrey, and M Sighireanu. *Introduction à E-LOTOS*, pages 213–252. Hermès Science, 2001.

[93] D E Long. *Model Checking, Abstraction and Compositional Verication.* PhD thesis, CMU School or Computer Science, 1993.

[94] N Lynch, R Segala, and F Vaandrager. Hybrid i/o automata. *Information and Computation*, 185(1):185–157, 2003.

[95] R Mateescu and M Sighireanu. Efficient on-the-fly model-checking for regular alternation-free μ-calculus. *Science of Computer Programming*, 46:255–281, 2003.

[96] Yael Meller, Orna Grumberg, and Sharon Shoham. A framework for compositional verification of multi-valued systems via abstraction-refinement. In *ATVA '09: Proceedings of the 7th International Sym-*

posium on Automated Technology for Verification and Analysis, pages 271–288, Berlin, Heidelberg, 2009. Springer-Verlag.

[97] Stephan Merz. Rules for abstraction. In R. K. Shyamasundar and K. Ueda, editors, *Advances in Computing Science—ASIAN'97*, volume 1345 of *Lecture Notes in Computer Science*, pages 32–45, Kathmandu, Nepal, December 1997. Springer-Verlag.

[98] R Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[99] J Misra and K M Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering SE-7*, 4:417–426, 1981.

[100] Wonhong Nam, P. Madhusudan, and Rajeev Alur. Automatic symbolic compositional verification by learning assumptions. *Form. Methods Syst. Des.*, 32(3):207–234, 2008.

[101] P W O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375:271–307, 2007.

[102] P W O'Hearn, J Reynolds, and H Yang. *Local reasoning about programs that alter data structures*, volume 2142 of *LNCS*, pages 1–19. Springer, 2001.

[103] Corina S. Pasareanu and Dimitra Giannakopoulou. Towards a compositional spin. In *SPIN*, pages 234–251, 2006.

[104] A Pnueli. *In transition for global to modular temporal reasoning about programs*, volume 13 of *NATO ASI*, chapter 5, pages 123–144. Springer, 1984.

[105] Junyan Qian, Lingzhong Zhao, Guoyong Cai, and Tianlong Gu. Formula-dependent abstraction for ctl model checking. In *ICCSA '08: Proceedings of the international conference on Computational Science and Its Applications, Part II*, pages 1035–1048, Berlin, Heidelberg, 2008. Springer-Verlag.

[106] Alexander Rabinovich. On compositionality and its limitations. *ACM Trans. Comput. Logic*, 8(1):4, 2007.

[107] Ramzi Ben Salah, Marius Bozga, and Oded Maler. On timed components and their abstraction. In *SAVCBS '07: Proceedings of the 2007 conference on Specification and verification of component-based systems*, pages 63–71, New York, NY, USA, 2007. ACM.

[108] Natasha Sharygina, Stefano Tonetta, and Aliaksei Tsitovich. An abstraction refinement approach combining precise and approximated techniques for efficient program verification: abstract for the invited talk. In *SAVCBS '09: Proceedings of the 8th international workshop on Specification and verification of component-based systems*, pages 35–36, New York, NY, USA, 2009. ACM.

[109] UPPAAL Team. Uppaal 3.4.8 help system, 2005.

[110] V Vafeidis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2007.

[111] V Vafeidis and M Parkinson. A marriage of rely/guarantee and separation logic. *Technical Report UCAM-CL-TR-687, University of Cambridge*, 2007.

[112] B Wachter and B Westphal. *The Spotlight Principle: On Combining Process-Summarizing State Abstractions*. Number 4349 in LNCS. Springer-Verlag, 2007.

[113] Chao Wang, Hyondeuk Kim, and Aarti Gupta. Hybrid cegar: combining variable hiding and predicate abstraction. In *ICCAD '07: Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design*, pages 310–317, Piscataway, NJ, USA, 2007. IEEE Press.

[114] P Wolper. The meaning of "formal": from weak to strong formal methods. *Springer International Journal of Software Tools for Technology Transfer*, 1(1-2):6–8, 1997.

[115] Qiwen Xu, Willem-Paul de Roever, and Jifeng He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9(2):149–174, 1997.

[116] Haiqiong Yao and Hao Zheng. Automated interface refinement for compositional verification. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 28(3):433–446, 2009.

Appendix A

Glossary of Symbols

Here are summarised the symbols and notations introduced in the body of the thesis. Here they are grouped by subject.

CTL*, Temporal Logic Syntax (§2.1.1)

$\varphi, \psi, \rho, \chi, ...$: state, path formulae;
$\bigcirc\varphi$: φ true next;
$\Diamond\varphi$: φ now or eventually;
$\Box\varphi$: φ true from now on;
$\varphi\mathcal{U}\psi$: ψ eventually; meanwhile φ;
$\varphi\mathcal{W}\psi$: φ now and until ψ;
$\varphi\mathcal{R}\psi$: ψ until φ first true;
$\forall\rho$: for all paths ρ ...
$\exists\rho$: for some path ρ ...

Temporal Logic Semantics (§2.1.2)

$\langle S, R, L\rangle$: a Kripke structure;
$L(s)$: the set of atomic formulae true at $s \in S$;
$\langle s_0, s_1, s_2, ...\rangle$ or $\overrightarrow{s}$: a path in the Kripke structure ($(s_i, s_{i+1}) \in R$);
$s \models \varphi$: φ is true at state s ($s \in S$);
$\langle \Sigma, \sigma^0, A, \rightarrow\rangle$: a labelled transition system with initial state σ^0;
$\sigma_1 \xrightarrow{\lambda} \sigma_2$: transition from σ_1 to σ_2 labelled λ;
$\sigma_0 \xrightarrow{\lambda_0} \sigma_1 \xrightarrow{\lambda_1} \sigma_2 \xrightarrow{\lambda_2} ...$: a run of the LTS;

Timed Temporal Logics (§2.1)

$\Diamond_{\leq m}\varphi$: parametrised $\Diamond$: φ will be true in up to m time units;
$\Box_{\leq m}\varphi$: parametrised $\Box$: φ will be true for at least m time units;
$\varphi\mathcal{U}_{<m}\psi$: ψ will be true less than m time units from now and in the mean time, φ is true;

μ-Calculus (§2.1.3)

$\wp(S)$: set of predicates over states (power set of S);
$f : \wp(S) \to \wp(S)$: a (monotonic) function;
$\mu X.f(X)$: least fixed point of f;
$\nu X.f(X)$: greatest fixed point of f;

Timed Automata (§2.2, §7.2)

$(L, l^0, A, \mathcal{H}, E, I)$: a timed automaton with locations set L, action label alphabet A, initial location l^0, clocks $\mathcal{H}$, edges E, invariant function I;
$I(l)$: invariant at location l;
ζ: a clock zone or constraint; a guard on an edge;
$\mathcal{Z}(\mathcal{H})$: the set of clock zones over $\mathcal{H}$;
$l \xrightarrow{\zeta,a,H} l'$: an edge from l to l' with guard ζ, label a, and set H of clocks to reset;
$\mathbb{A}_1 \parallel \mathbb{A}_2$: composite of timed automata $\mathbb{A}_1$, $\mathbb{A}_2$;(§2.2.3)

Timed Automata Semantics (§2.2.2)

$\langle \Sigma, \sigma^0, A \cup \mathbb{R}, \to \rangle$: timed transition system of the automaton;
$\Sigma \triangleq L \times \mathbb{R}^{\mathcal{H}}$;
$\sigma = (l, v) \in \Sigma$: a timed state at location l, with clock valuation v;
$a(\in A)$: a discrete action transition;
$t(\in \mathbb{R})$: a time-elapse transition;
$v + t$: clock valuation which gives h the value $v(h) + t$;
$v[h := 0]$: clock valuation like v except h has value 0;

UPPAAL Semantics and Compositionality (§5.2.2, §7.5.1, §7.5.2)

$(\overrightarrow{l}, v)$: a timed state: $\overrightarrow{l} = (l_1, ..., l_n)$ is a vector of component process "states" (locations) and v a valuation of variables;

$(\overrightarrow{l}, v) \rightarrow (\overrightarrow{l}', v')$: a transition;

$(\overrightarrow{l}, v) \xrightarrow{d} (\overrightarrow{l}, v)$: a time-passage transition $(d \in \mathbb{R}, d \geq 0)$;

$l_k \xrightarrow{\zeta,\lambda} l'_k$: an edge of one component with guard ζ, update λ and no synchronisation label gives rise to an internal (interleaving) transition;

$l_j \xrightarrow{\zeta_j,c!,\lambda_j} l'_j$ and $l_k \xrightarrow{\zeta_k,c?,\lambda_k} l'_k$: a pair of component edges with complementary synchronisation labels $c!, c?$ on channel c gives rise to a binary synchronisation transition;

$l_j \xrightarrow{\zeta_j,c!,\lambda_j} l'_j$ and in m (possibly 0) component processes $k_1, ..., k_m$ edges $l_{k_p} \xrightarrow{\zeta_{k_p},c?,\lambda_{k_p}} l'_{k_p}$ $(p = 1, ..., m)$: gives rise to the broadcast synchronisation on broadcast channel c;

$(v, (l_i^k)_{i=1...n})$: a timed state of an UPPAAL system with valuation v, vector of locations $(l_i^k)_{i=1...n}$ (§7.5.1);

$v + t$: advance of timed valuation v by time t;

$v[\lambda]$: a timed valuation updated by λ;

$v \models \bigwedge_{i=1}^{m} I_i(l_i^k)$: valuation v satisfies all the invariants of $(l_i^k)_{i=1...n}$;

$v \models \zeta_i \wedge \zeta_{j_1} \wedge ... \wedge \zeta_{j_m}$: v satisfies several guards;

$\prod_{i=1}^{n} \mathbb{A}_i$ or $\mathbb{A}_1 \parallel ... \parallel \mathbb{A}_n$: a parallel composite of UPPAAL process automata;

$\mathbb{A} \preceq \mathbb{B}$: trace inclusion;

Symbolic Model Checking with OBDD (§2.3.2)

$\mathbf{2}$: the two-element Boolean algebra;

$f_v : \mathbf{2}^n \rightarrow \mathbf{2}$: the boolean function determined at vertex v of a binary decision diagram;

f, $\neg f$, $f \vee g$, $f \wedge g$, $\exists v f$, $\forall v f$: QBF formulae (v a propositional variable);

$\sigma \models f$: with truth assignment σ, t evaluates to 1 ($true$);

Model Checking with Timed automata (§2.3.5)

$v \simeq v'$: clock-region-equivalence of two clock valuations;

$\zeta + t$: clock zone ζ advanced by t;

$\zeta[H := 0]$: clock zone like ζ but with clocks in H reset;

Bisimulation and simulation (§2.4)

$s_1 \simeq s_2$: strong equivalence of states of a labelled transition system;
$(S_1, S_1^0, A, \rightarrow_1) \preceq (S_2, S_2^0, A, \rightarrow_2)$: transition system 1 is simulated by transition system 2.

bCANdle Data Model (§4.1)

V, Var, Ω, Γ: (sets of) data values, variables, operation names, predicate names;
$D = (type, op, pred, val)$: a data environment over Var, Ω, Γ;
$D' = D[x := v]$: D' like D except $D'.x = v$;
$D \xrightarrow{\omega}_d D'$: data model transition relation: D evolves to D' when ω has run;
$D \models \gamma$: predicate γ is true in data model D;

bCANdle Network Model (§4.2)

I, V: (sets of) message identifiers (type), data values;
(i, v) or $i.v$ $(i \in I, v \in V)$: a message;
$i \preccurlyeq i'$, $m = (i, v) \preccurlyeq (i', v') = m'$: priority ordering of messages;
$m \prec m'$: strict priority ordering;
$\delta(m) = (l, u, L, U)$: time bounds of preacceptance phase (l, u) and postacceptance phase (L, U);
$\downarrow$: channel state: free;
$\overset{t_1,t_2}{\rightsquigarrow} m$: channel in preacceptance phase of message m with time to completion bounded by t_1, t_2;
$\uparrow m$: channel at point of accepting message m;
$m \overset{t_1,t_2}{\rightsquigarrow}$: channel in postacceptance phase of message m;
u: a message queue; $m : u$: a message queue with m at the head;
$u \leftharpoonup i.v$: insertion of message $i.v$ in queue u;
N: a network is an indexed set of channels $N_k : k \in K$;
$N_k = (s, u)$: channel k is in state s with message queue $= u$;
$k \rightsquigarrow i.v$: a network action: enter preacceptance phase of $i.v$ on channel k;
$k \uparrow i.v$: a network action: acceptance of $i.v$ on channel k;
$i.v \rightsquigarrow k$: a network action: enter postacceptance phase of $i.v$ on channel k;

$k\downarrow$: network action: channel k becomes free;
$N \xrightarrow{a}_n N'$: network model transition relation: N evolves to N' upon action a;
$(s, u) + t$ (where $t \in \mathbb{R}$): effect of time elapse on channel state;
$N \xrightarrow{t}_n N'$: network model transition relation: N evolves to N' upon time passage t;

bCANdle Process Term Syntax (§3.2.2, §4.3)

$k!i.x$: sending a message (payload x) of type i on channel k;
$k?i.x$: receiving a message of type i on channel k into x;
$[\omega : t_1, t_2]$: a data operation ω with time bounds t_1, t_2;
$\gamma \to P$: a process term P guarded by a predicate γ;
$P; Q$, $P + Q$, $P[> Q$, $P|Q$: sequence, nondeterministic choice, interrupt, parallel composite of process terms P, Q;
X: a process variable;
$recX.P$: the recursion construct;
(P, N, D): a bCANdle system;
$\surd$: "ideal" process term, denoting completion;
$(P, N, D) \xrightarrow{\lambda} (P', N', D')$: a bCANdle transition;
λ may be a network action λ_n, a time passage t, or a process action arising from a data model transition ω, validation of a predicate γ, a send action $k!i.v$ or a receive action $k?i.v$;

Compositional Model of bCANdle (§5.4)

$k!i.x$, $k?i.x$, $[\omega : t_1, t_2]^h$, X: atomic clocked process terms,
$\gamma \to \hat{Q}$, $recX.\hat{Q}$, $\hat{Q}; \hat{Q}'$, $\hat{Q} + \hat{Q}'$, $\hat{Q}[> \hat{Q}'$: clocked process terms (§5.4.1);
$\mathbb{A}(\hat{P}_0, D_0)$: the timed automaton modelling a clocked process term with data environment (§5.4.2);
$(\hat{Q}, D) \xrightarrow{\zeta,\lambda,H} (\hat{Q}', D')$: an edge with guard ζ, label λ, update set H;

bCANdle as a Parallel Product (§5.5)

$\|_{i=1}^n \mathbb{A}(\hat{P}_i, D_i) \parallel (\|_{k \in K} N_k)$: a parallel product with timed transition system strongly equivalent to that (P, N, D) (Theorem 5.5.1);

$((\hat{Q}_i, D_{Qi})_{i=1...m}, (\eta_k)_{k\in K}, v)$: a general state of the timed transition system ($(\eta_k)_{k\in K}$ is a vector of network channel automaton locations, v a valuation; §5.5.2);
$\texttt{unclk}(\hat{Q})$: the `unclk` function on clocked process terms;
$\texttt{age}(\hat{Q}, v)$: the `age` function on clocked process terms;

CAN Without Shared Variables (§6.1, figure 6.1)

$\texttt{nMsg}[\mu, \nu]$: a two-index family (μ ranges over message identifier/types, ν over payload values) of UPPAAL urgent synchronisation channels;
$\texttt{M2S}[\mu, \nu]$, $\texttt{PreAcc}[\mu, \nu]$, $\texttt{PostAcc}[\mu, \nu]$: families of UPPAAL locations;
$\texttt{dMsg}[\mu, \nu]$: a family of UPPAAL broadcast synchronisation channels;

Timed I/O Automata (KLSV Structures) (§3.4.3, §7.2.1)

$\mathbb{A} = (X, Q, \Theta, E, H, \mathcal{D}, \mathcal{T})$: a KLSV structure with variables X, states Q, external and internal action labels E, H, discrete action relation $\mathcal{D} \subseteq Q \times (E \cup H) \times Q$, and trajectories $\mathcal{T}$;
$\tau : J \to Q$: a trajectory (domain of J is of form $[0, b]$ or $[0, b)$ or $[0, \infty)$);
$\tau : [0, b] \to Q$: a closed trajectory;
$\tau : [0, 0] \to Q$: a point trajectory;
$\tau \leq \tau'$: τ is a prefix of τ';
$\tau \trianglerighteq t$ ($t \in dom(\tau)$: suffix of a trajectory;
$\tau_0 \frown \tau_1 \frown \tau_2 \frown ...$: concatenation of trajectories;
$\tau_0 a_1 \tau_1 a_2 \tau_2 a_3 ...$: an infinite execution fragment ($\tau_i \in \mathcal{T}$, $a_i \in E \cup H$);
$\tau_0 a_1 \tau_1 a_2 \tau_2 a_3 ... \tau_n$: a finite infinite execution fragment;
$\mathbb{A}_1 \parallel \mathbb{A}_2$: composition (§7.2.3);
$\mathbb{A}_1 \preceq \mathbb{A}_2$: implementation, trace inclusion(§7.3.2);

Appendix B

Bisimulation Proof Details

Here are details of the proof that the mapping (5.5) is a strong bisimulation up to strong equivalence of bCANdle systems with product timed automata of the type constructed in section 5.4.

To see this one needs to check, for $((\hat{Q}_i, D_{Qi})_{i=1\ldots m}, (\eta_k)_{k\in K}, v)$ a general timed state of the automaton, and $(|_{i=1}^n \mathsf{age}(\hat{Q}_i, v), (\mathsf{age}(\eta_k, v))_{k\in K}, \sqcup_{i=1}^n D_{Qi})$ the bCANdle system to which it maps,

1. $((\hat{Q}_i, D_{Qi})_{i=1\ldots m}, (\eta_k)_{k\in K}, v) \approx (|_{i=1}^n \mathsf{age}(\hat{Q}_i, v), (\mathsf{age}(\eta_k, v))_{k\in K}, \sqcup_{i=1}^n D_{Qi})$;

2. For every transition λ from $(|_{i=1}^n \mathsf{age}(\hat{Q}_i, v), (\mathsf{age}(\eta_k, v))_{k\in K}, \sqcup_{i=1}^n D_{Qi})$ in the bCANdle timed transition system there is a transition with the same label λ from $((\hat{Q}_i, D_{Qi})_{i=1\ldots m}, (\eta_k)_{k\in K}, v)$, and the targets are context equivalent to states related by the mapping;

3. For every transition λ from $((\hat{Q}_i, D_{Qi})_{i=1\ldots m}, (\eta_k)_{k\in K}, v)$ there is a transition with the same label λ from $(|_{i=1}^n \mathsf{age}(\hat{Q}_i, v), (\mathsf{age}(\eta_k, v))_{k\in K}, \sqcup_{i=1}^n D_{Qi})$, and the targets are context equivalent to states related by the mapping.

The first of these requirements is met by definition of context equivalence.

The second is established by considering a number cases and using induction on the computation of $|_{i=1}^n \mathsf{age}(\hat{Q}_i, v)$, as shown below.

The third is established in a way symmetrical with the second, again by induction on the computation of $|_{i=1}^n \mathsf{age}(\hat{Q}_i, v)$. Further details are given

below.

B.1 Lemmas

To assist with this a number of lemmas a presented. Versions of these appear in Kendall [79], appendix B where they appear as lemma B1 and lemmas B3 – B8; they are adapted to work with the compositional timed-automaton formalism for bCANdle presented in this thesis.

B.1.1 Lemma

If (P_1, N, D_1), (P_2, N, D_2) are two bCANdle systems with the *same* network state N and $(P_1, N, D_1) \xrightarrow{\lambda} (P_1', N', D_1')$ and $(P_2, N, D_2) \xrightarrow{\lambda} (P_2', N'', D_2')$ are bCANdle timed transitions from these systems with the same label λ then $N' = N''$, $D_i' = D_i$ $(i = 1, 2)$: the target network states agreed and data environments are unchanged.

If, in addition, $\lambda \in A_n$, a discrete network action, then $P_i' \equiv P_i$.

This is a straightforward induction on the rule §4.3 that defines the transitions (behaviour) of bCANdle systems. This lemma needs no adaptation from [79].

B.1.2 Lemma

Let $\mathbb{A}(\hat{P}_0, D_0)$ as defined in §5.4.2 denote the timed automaton of a clocked process term with its associated data environment. Let v be a clock valuation defined on the clocks of the automaton, and H a set of clock variables $\subseteq$ the domain of v. Let $(\hat{Q}, D)$ denote a location of $\mathbb{A}(\hat{P}_0, D_0)$. If $v \models I(\hat{Q}, D)$ then $v[H := 0] \models I(\hat{Q}, D)$.

If a location invariant is satisfied by a clock valuation, it is still satisfied when some additional clocks are reset.

This is easily proven by induction on the definition of I in $\mathbb{A}(\hat{P}_0, D_0)$ (see §5.4.2.1). Remember the "urgent" clock h^u is reset on *every* edge.

B.1.3 Lemma

Let $(\hat{Q}, D)$ denote a location of $\mathbb{A}(\hat{P}_0, D_0)$ as in lemma B.1.2 above. Let v be a clock valuation defined on the clocks of the automaton, and H a set of clock variables $\subseteq$ the domain of v. Provided $h^u \in H$, for any data environment D' *compatible* (ref §4.1) with D, $v \models I(\hat{Q}, D)$ implies $v[H := 0] \models I(\hat{Q}, D')$.

Resetting some additional clocks satisfies the invariant in any compatible environment, provided the urgent clock is reset.

Again this is easily seen by induction on the definition of I in $\mathbb{A}(\hat{P}_0, D_0)$.

B.1.4 Lemma

Let $\hat{Q}$ denote a clocked process term. Let v be a clock valuation defined on the clocks of $\hat{Q}$, and H a set of clock variables $\subseteq$ the domain of v. Recall from §5.4.2.2 the set $H_{\hat{Q}}$ of *initial clock variables* of $\hat{Q}$. Provided $h^u \in H$ and $H_{\hat{Q}} \subseteq H$, for any data environment D, $v[H := 0] \models I(\hat{Q}, D)$.

Again this follows by induction on the definition of I.

B.1.5 Lemma

1. Let K denote a set of network channel identifiers, η_k $(k \in K)$ a vector of network channel automata locations, and v a clock valuation defined on all the clocks of the η_k; let $\mathsf{age}(\eta, v)$ be an abbreviation for the network environment defined by $(\mathsf{age}(\eta_k, v))_{k \in K}$.

   If v, v' are two such clock valuations which agree on all the clocks in the η_k, then $\mathsf{age}(\eta, v) = \mathsf{age}(\eta, v')$ – that is $(\forall k \in K)\mathsf{age}(\eta_k, v) = \mathsf{age}(\eta_k, v')$.

2. If $\hat{Q}$ denotes a clocked process term and v, v' are two clock valuations which agree on all the clocks in $H_{\hat{Q}}$, then $\mathsf{age}(\hat{Q}, v) = \mathsf{age}(\hat{Q}, v')$.

B.1.6 Lemma

Let $\hat{Q}$ denote a clocked process term and v a valuation defined on all the clocks of $\hat{Q}$, and H a set of clock variables $\subseteq$ the domain of v. Let $H_{\hat{Q}}$ denote

the set of initial clock variables of $\hat{Q}$ (§5.4.2.2) and $\texttt{unclk}(\hat{Q})$ be defined as in 5.5.1.

If $H_{\hat{Q}} \subseteq H$, then $\texttt{age}(\hat{Q}, v[H := 0]) \simeq \texttt{unclk}(\hat{Q})$.

This proven by induction on the steps in the definition of $\texttt{age}(.)$.

B.1.7 Lemma

Let $\eta = (\eta_k)_{k \in K}$ denote a vector of network channel automata locations, and v a clock valuation defined on all its clocks, as above. Let $I(\eta) \triangleq \bigwedge_{k \in K}(\eta_k)$. Let t be a time increment. Let $\texttt{tcp}(.)$ denote the maximum time progress function on bCANdle network channel states defined in 4.2.

If $v + t \models I(\eta)$ then $t \leq \texttt{tcp}(\texttt{age}(\eta, v))$; i.e., $(\forall k \in K) t \leq \texttt{tcp}(\texttt{age}(\eta_k, v))$. If a clock valuation can be incremented by t while all the network channel invariants continue to be satisfied, then the corresponding aged (bCANdle) network allows passage of t time units.

This assertion follows in a straightforward way from the definition of $\texttt{tcp}(.)$ and the bCANdle network channel state rule (N5).

B.1.8 Lemma

This is Proposition B4 of [79]. If $(P, N, D) \simeq (\surd, N, D)$ as bCANdle systems, then $P \equiv \surd$. Only $\surd$ behaves like $\surd$.

B.2 The Bisimulation – Details

Further details of the bisimulation now follow. The sketches below contain some abuse of notation: for instance k may denote a particular channel and at the same time be employed as a dummy index over K, and v may denote a valuation, part of a timed state in a timed transition system, and also value payload of a message, as in $k!i.v$. The context should make the meanings clear in these cases. A vector of channel automaton locations, $(\eta_k)_{k \in K}$ figures frequently as part of the automaton state: it will be abbreviated simply η, and $\texttt{age}(\eta, v)$ will in this context abbreviate the vector $(\texttt{age}(\eta_k, v))_{k \in K}$. Similarly,

η' abbreviates $(\eta'_k)_{k\in K}$ and so forth. The symbol $\mathbb{R}$ when used to denote *time* values will strictly mean $\{t \in \mathbb{R} : t \geq 0\}$.

The relation we aim to establish as a bisimulation is that given by the mapping

$$((\hat{Q}_i, D_{Qi})_{i=1\ldots n}, (\eta_k)_{k\in K}, v) \overset{(5.5)}{\longmapsto} (|_{i=1}^{n}\mathsf{age}(\hat{Q}_i, v), (\mathsf{age}(\eta_k, v))_{k\in K}, \sqcup_{i=1}^{n} D_{Qi})$$

which maps a timed state of the product automaton – a vector of $\mathbb{A}(\hat{Q}_i, D_{Qi})$ and η_k (channel automaton) locations and a clock valuation v – to a bCANdle system constructed from these elements using the age function.

It is straightforward to check that the right hand side is a well-formed bCANdle system and we have already noted that $((\hat{Q}_i, D_{Qi})_{i=1\ldots n}, (\eta_k)_{k\in K}, v) \approx (|_{i=1}^{n}\mathsf{age}(\hat{Q}_i, v), (\mathsf{age}(\eta_k, v))_{k\in K}, \sqcup_{i=1}^{n} D_{Qi})$ – the two sides are are context-equivalent, by construction of the extended context-equivalence relation.

It remains to check the proposed bisimulation relation $\overset{(5.5)}{\longmapsto}$ respects timed transitions in each direction, up to strong equivalence ($\simeq$).

B.3 A Transition in bCANdle is Simulated in the TA Formalism

First, it is required that for every transition $(|_{i=1}^{m}\mathsf{age}(\hat{Q}_i, v), (\mathsf{age}(\eta_k, v))_{k\in K}, \sqcup_{i=1}^{m} D_{Qi}) \xrightarrow{\lambda} (Q', N', D')$ in the bCANdle timed transition system there is a transition with the same label λ from $((\hat{Q}_i, D_{Qi})_{i=1\ldots m}, (\eta_k)_{k\in K}, v)$, and the targets are context equivalent to states related by the mapping.

Since the age function is defined recursively on clocked process terms, this will be proven by induction on the definition of age. To this end, it suits us to prove a slightly stronger property: for every pair of timed states $((\hat{Q}_i, D_{Qi})_{i=1\ldots n}, (\eta_k)_{k\in K}, v) \overset{(5.5)}{\longmapsto} (|_{i=1}^{n}\mathsf{age}(\hat{Q}_i, v), (\mathsf{age}(\eta_k, v))_{k\in K}, \sqcup_{i=1}^{n} D_{Qi})$ related by the mapping, for every transition $(|_{i=1}^{m}\mathsf{age}(\hat{Q}_i, v), (\mathsf{age}(\eta_k, v))_{k\in K}, \sqcup_{i=1}^{m} D_{Qi}) \xrightarrow{\lambda} (Q', N', D')$ in the bCANdle

timed transition system there is a transition with the same label $((\hat{Q}_i, D_{Qi})_{i=1...m}, (\eta_k)_{k\in K}, v) \xrightarrow{\lambda} ((\hat{Q}'_i, D'_{Qi})_{i=1...m'}, (\eta'_k)_{k\in K}, v')$, where for some clocked bCANdle terms, $\hat{Q}''_i \simeq \hat{Q}'_i, (i = 1...m')$,
$((\hat{Q}''_i, D'_{Qi})_{i=1...m'}, (\eta'_k)_{k\in K}, v) \overset{(5.5)}{\longmapsto} (|_{i=1}^{m'}\mathsf{age}(\hat{Q}''_i, v), (\mathsf{age}(\eta'_k, v'))_{k\in K}, \sqcup_{i=1}^{m'} D'_{Qi})$
and $|_{i=1}^{m'}\mathsf{age}(\hat{Q}''_i, v') \simeq Q'$ and $N' = \mathsf{age}(\eta', v')$ and $D' = \sqcup_{i=1}^{m'} D'_{Qi}$.

Proof Here are the cases needing to be considered:

1. $(\hat{Q}_i, D_{Qi})_{i=1...m}$ is a single term $(k!i.x, D_{Q1})$ where $k \in K, i \in I, x \in Var$. Then $Q_1 = \mathsf{age}(\hat{Q}_1, v) = k!i.x$. λ can be one of three things:

   (a) λ is $k!i.v$ arising by rule (Snd1) of §4.3, where $v = D_{Q1}.x$.
   Then $N' = \mathsf{age}(\eta, v))[k := (s, u \leftarrow\!\!\!\rho\ i.v)]$ and $D' = D_{Q1},\ Q' = \surd$.
   By rule (C_Snd) (§5.4.2.2) there is an edge $(k!i.x, D_{Q1}) \overset{1,k!i.v,\{h^u\}}{\longrightarrow} (\surd, D_{Q1})$ in $\mathbb{A}(\hat{Q}_1, D_{Q1})$ which synchronises with one of the transitions in η_k which inserts $i.v$ into the queue for channel k. The result is a transition
   $((\hat{Q}_1, D_{Q1}), \eta, v) \xrightarrow{k!i.v} ((\surd, D_{Q1}), (\eta'_k)_{k\in K}, v[h^u := 0])$ where the $\eta'_k (k \in K)$ are like η_k except for the particular k which is the subject of rule (Snd1) for which η'_k is η_k with $i.v$ inserted in the queue (e.g. $Q[i] := v$).

   It remains to check that
   $((\surd, D_{Q1}), (\eta'_k)_{k\in K}, v[h^u := 0]) \longmapsto (Q', N', D')$. We already have $Q' = \surd, D' = D_{Q1}$. Also, $\mathsf{age}(\eta', v) = N'$.

   (b) λ arises from a network transition $N \xrightarrow{\lambda_n}_n N'$ on a particular channel k, via §4.3(Snd 2). This is one of

   - $N \xrightarrow{k\rightsquigarrow m}_n N' = N[k := (\overset{l(m),u(m)}{\rightsquigarrow} m, u)]$ where $N_k = (\downarrow, m : u)$;
   - $N \xrightarrow{k\uparrow m}_n N' = N[k := (\uparrow m, u)]$ where $N_k = (\overset{0,t}{\rightsquigarrow} m, u)$ for some t;
   - $N \xrightarrow{m\rightsquigarrow k}_n N' = N[k := (m \overset{L(m),U(m)}{\rightsquigarrow}, u)]$ where $N_k = (\uparrow m, u)$;
   - $N \xrightarrow{k\downarrow}_n N' = N[k := (\downarrow, u)]$ where $N_k = (m \overset{0,t}{\rightsquigarrow}, u)$ for some t;

In each of these cases we have $\mathtt{age}(k!i.x, D_{Q1}) = k!i.x$ and $N = \mathtt{age}(\eta, v)$ and $(k!i.x, N, D_{Q1}) \xrightarrow{\lambda} (k!i.x, N', D_{Q1})$. The corresponding transition in the timed-automaton formalism of §5.4.2 is
$((\hat{Q}_1, D_{Q1}), \eta, v) \xrightarrow{\lambda} ((\hat{Q}_1, D_{Q1}), (\eta'_k)_{k\in K}, v[h^u := 0])$ where λ is a transition on the particular channel k:

$$\begin{aligned}
\eta_k = \mathtt{Free} &\Rightarrow \eta'_k = \mathtt{PreAcc}\\
\eta_k = \mathtt{PreAcc} &\Rightarrow \eta'_k = \mathtt{Acc}\\
\eta_k = \mathtt{Acc} &\Rightarrow \eta'_k = \mathtt{PostAcc}\\
\eta_k = \mathtt{PostAcc} &\Rightarrow \eta'_k = \mathtt{Free}
\end{aligned}$$

It needs to be checked that $\mathtt{age}(\eta', v) = (\mathtt{age}(\eta'_k, v))_{k\in K} = N'$ but the guards and invariants of the channel automaton have been chosen to ensure this.

(c) $\lambda = 0 : (k!i.x, N, D_{Q1}) \longrightarrow (k!i.x, N, D_{Q1})$ by §4.3(Snd 3) with $N = \mathtt{age}(\eta, v)$. This is mirrored in the timed-automaton formalism by the transition
$((k!i.x, D_{Q1}), \eta, v) \xrightarrow{0} ((k!i.x, D_{Q1}), \eta, v)$ which is enabled assuming v satisfies the invariants of the source state. Also,
$((k!i.x, D_{Q1}), \eta, v) \longmapsto (k!i.x, N, D_{Q1})$ under the mapping (5.5) because $N = \mathtt{age}(\eta, v)$ by hypothesis.

2. $(\hat{Q}_i, D_{Qi})_{i=1\ldots m}$ is a single term $(k?i.x, D_{Q1})$ where $k \in K, i \in I, x \in Var$. $Q_1 = \mathtt{age}(\hat{Q}_1, v) = k?i.x$. Again, there are three possibilities for $(k?i.x, \mathtt{age}(\eta, v), D_{Q1}) \xrightarrow{\lambda} (Q', N', D')$:

(a) $\lambda = k?i.v$ by rule §4.3(Rcv 1). By hypothesis, $N_k = (\uparrow i.v, u)$ (so $\eta_k = \mathtt{Acc}$), $Q' = \surd, N' = \mathtt{age}(\eta, v), D' = D_{Q1}[x := v]$.

To obtain the corresponding transition in the timed-automaton formalism, C_Rcv (§5.4.2.2) provides an edge
$(k?i.x, D_{Q1}) \xrightarrow{1, k?i.v, h^u} (\surd, D_{Q1}[x := v])$ in which $k!i.v$ receives a broadcast synchronisation with the channel automaton transition

leaving location `Acc`. This results in the transition
$((k?i.x, D_{Q1}), \eta, v) \xrightarrow{k?i.v} ((\surd, D_{Q1}[x := v]), \eta, v[h^u := 0])$.

(b) λ is a network transition and bCANdle rule §4.3(Recv 2) is employed. Thus, $\lambda = \lambda_n$ is a network transition on a channel $k' \neq k$, or else on channel k but with N_k *not* at acceptance point. The construction of a parallel transition in the timed-automaton formalism is as in the (Snd 2) case.

(c) $\lambda = t \in \mathbb{R}$, a time delay to which §4.3 rule (Rcv 3) applies. The parallel transition in the timed-automaton formalism is
$((k?i.x, D_{Q1}), \eta, v) \xrightarrow{t} ((k?i.x, D_{Q1}), \eta, v + t)$.

3. $(\hat{Q}_i, D_{Qi})_{i=1...m}$ is a single term $([\omega : t_1, t_2]^h, D_{Q1})$. $\mathtt{age}([\omega : t_1, t_2]^h) = [\omega : t_1 \dot{-} v(h), t_2 \dot{-} v(h)]$ and so the mapping/relation (5.5) takes the form
$(([\omega : t_1, t_2]^h, D_{Q1}), \eta, v) \longmapsto ([\omega : t_1 \dot{-} v(h), t_2 \dot{-} v(h)], \mathtt{age}(\eta, v), D_{Q1})$.
There are three sub-cases for
$([\omega : t_1 \dot{-} v(h), t_2 \dot{-} v(h)], \mathtt{age}(\eta, v), D_{Q1}) \xrightarrow{\lambda} (Q', N', D')$:

(a) $([\omega : t_1 \dot{-} v(h), t_2 \dot{-} v(h)], N_1, D_{Q1}) \xrightarrow{\omega} (\surd, N_1, D'_{Q1})$ by §4.3 rule (Comp 1), where $t_1 \dot{-} v(h) = 0$, $N_1 = \mathtt{age}(\eta, v)$, and D'_{Q1} is the data model state after the operation ω, as determined by §5.4.2.2 rule (C_Cmp).

In this case, construct the corresponding automaton transition
$(([\omega : t_1, t_2]^h, D_{Q1}), \eta, v) \xrightarrow{\omega} ((\surd, D'_{Q1}), \eta, v[h^u := 0])$

(b) λ is a network action, the transition arising via §4.3(Comp 2). This case is handled similarly to the (Snd 2), (Rcv 2) cases.

(c) $\lambda = t \in \mathbb{R}$ via §4.3(Comp 3). This case is similar to the (Rcv 3) case.

4. $(\hat{Q}_i, D_{Qi})_{i=1...m}$ is a single term $(\hat{Q}_1, D_{Q1})$ where $\hat{Q}_1 = \gamma \rightarrow \hat{Q}$ and $\gamma \in \Gamma$ is a predicate. The mapping relation (5.5) takes the form $((\hat{Q}_1, D_{Q1}), \eta, v) \longmapsto (\mathtt{age}(\hat{Q}_1, v), \mathtt{age}(\eta, v), D_{Q1})$; and $\mathtt{age}(\hat{Q}_1, v) = \gamma \rightarrow \mathtt{unclk}(\hat{Q})$.

(a) One sub-case is that in which $D_{Q1} \models \gamma$ and a transition $\lambda = \gamma$ is inferred from §4.3 rule (Gu 1):
$(\gamma \rightarrow \texttt{unclk}(\hat{Q}), \texttt{age}(\eta, v), D_{Q1}) \xrightarrow{\gamma} (\texttt{unclk}(\hat{Q}), \texttt{age}(\eta, v), D_{Q1})$.
In this case rule §5.4.2.2(C_Gd) gives in the automaton formalism an edge $(\gamma \rightarrow \hat{Q}, D_{Q1}) \overset{1,\gamma,\{h^u\}\cup H_{\hat{Q}}}{\longrightarrow} (\hat{Q}, D_{Q1})$. Using lemmas B.1.2, B.1.4, one can check that $\mathrm{v}[\{h^u\} \cup H_Q := 0] \models I(\hat{Q}, D_{Q1})$ and infer a timed transition
$((\gamma \rightarrow \hat{Q}, D_{Q1}), \eta, v) \xrightarrow{\gamma} ((\hat{Q}, D_{Q1}), \eta, v[\{h^u\} \cup H_{\hat{Q}} := 0])$ and the case is completed by checking that by mapping (5.5),
$((\hat{Q}, D_{Q1}), \eta, v[\{h^u\} \cup H_{\hat{Q}} := 0]) \longmapsto (\texttt{unclk}(\hat{Q}), \texttt{age}(\eta, v), D_{Q1})$.

(b) $(\gamma \rightarrow \texttt{unclk}(\hat{Q}), \texttt{age}(\eta, v), D_{Q1}) \xrightarrow{\lambda_n} (\texttt{unclk}(\hat{Q}), N', D_{Q1})$ by §4.3 (Gu 2) where $\lambda_n : N = \texttt{age}(\eta, v) \longrightarrow N'$ is a discrete network action on some channel k. In this case $N' = (\texttt{age}(\eta'_{k'}, v))_{k' \in K} = \texttt{age}(\eta', v)$ for short. $\eta_{k'} = \eta'_{k'}$ for all $k' \neq k$ and η'_k is the channel automaton location following η_k. The rest of the details resemble the (Snd 2) case.

(c) $(\gamma \rightarrow \texttt{unclk}(\hat{Q}), \texttt{age}(\eta, v), D_{Q1}) \xrightarrow{t\in\mathbb{R}} (\texttt{unclk}(\hat{Q}), N', D_{Q1})$ by §4.3 (Gu 3) where $t = 0$ or $D_{Q1} \not\models \gamma$, and $N = \texttt{age}(\eta, v) \xrightarrow{t} N'$ is a time delay on the network. In these cases $N' = \texttt{age}(\eta', v+t)$. The case is completed in a fashion similar to the (Rcv 3), (Comp 3) cases.

5. $(\hat{Q}_i, D_{Qi})_{i=1\ldots m}$ is a single term $(\hat{Q}_1, D_{Q1})$ where $\hat{Q}_1 = \hat{Q}_{11}; \hat{Q}_{12}$, a sequence of two (clocked) process terms. The mapping relation (5.5) takes the form $((\hat{Q}_1, D_{Q1}), \eta, v) \longmapsto (\texttt{age}(\hat{Q}_1, v), \texttt{age}(\eta, v), D_{Q1})$; and $\texttt{age}(\hat{Q}_1, v) = \texttt{age}(\hat{Q}_{11}; \hat{Q}_{12}, v) = \texttt{age}(\hat{Q}_{11}, v); \texttt{unclk}(\hat{Q}_{12})$.

(a) The first sub-case is that in which there is a transition inferred by rule §4.3(Seq 1), $(\texttt{age}(\hat{Q}_{11}, v); \texttt{unclk}(\hat{Q}_{12}), \texttt{age}(\eta, v), D_{Q1}) \xrightarrow{\lambda} (Q'_{11}; \texttt{unclk}(\hat{Q}_{12}), N', D')$ from $(\texttt{age}(\hat{Q}_{11}, v), \texttt{age}(\eta, v), D_{Q1}) \xrightarrow{\lambda} (Q'_{11}, N', D')$, where Q_{11} is not $\surd$ and $\lambda \in A_p \cup A_n \cup \mathbb{R}$.

By induction hypothesis, there is a timed transition in the automaton formalism, $((\hat{Q}_{11}, D_{Q1}), \eta, v) \xrightarrow{\lambda} ((\hat{Q}'_{11}, D'_{Q1}), \eta', v')$ with

$\hat{Q}'_{11} \simeq \hat{Q}''_{11}$, $Q'_{11} \simeq Q''_{11}$ and $((\hat{Q}''_{11}, D'_{Q1}), \eta', v') \longmapsto (Q''_{11}, N', D')$ by (5.5).

There are three "sub-sub-cases" to consider, depending on the type of transition λ:

i. If $\lambda \in A_p$, a discrete process action, then there must be an automaton edge $(\hat{Q}_{11}, D_{Q1}) \xrightarrow{\zeta,\lambda,H} (\hat{Q}'_{11}, D'_{Q1})$ with $v \models \zeta$, $v' = v[H := 0] \models I(\hat{Q}'_{11}, D'_{Q1})$.
Now, §5.4.2.2 rule (C_Seq 1) implies an edge
$(\hat{Q}_{11}; \hat{Q}_{12}, D_{Q1}) \xrightarrow{\zeta,\lambda,H} (\hat{Q}'_{11}; \hat{Q}_{12}, D'_{Q1})$ with
$v \models \zeta$ and $v' = v[H := 0] \models I(\hat{Q}'_{11}; \hat{Q}_{12}, D'_{Q1})$.
From this can be inferred a timed transition
$((\hat{Q}_{11}; \hat{Q}_{12}, D_{Q1}), \eta, v) \xrightarrow{\lambda} ((\hat{Q}'_{11}; \hat{Q}_{12}, D'_{Q1}), \eta', v')$ where λ may synchronise with an edge of a channel automaton to make a change of a network state.
Since $((\hat{Q}''_{11}, D'_{Q1}), \eta', v') \longmapsto (Q''_{11}, N', D')$, $\hat{Q}'_{11} \simeq \hat{Q}''_{11}$, $Q'_{11} \simeq Q''_{11}$, it follows that
$\hat{Q}'_{11}; \hat{Q}_{12} \simeq \hat{Q}''_{11}; \hat{Q}_{12}$, $Q'_{11}; \texttt{unclk}(\hat{Q}_{12}) \simeq Q''_{11}; \texttt{unclk}(\hat{Q}_{12})$ and
$((\hat{Q}''_{11}; \hat{Q}_{12}, D'_{Q1}), \eta', v') \overset{(5.5)}{\longmapsto} (Q''_{11}; \texttt{unclk}(\hat{Q}_{12}), N', D')$ as required.

ii. If $\lambda \in A_n$, a discrete network action, then λ arises from an edge of a channel automaton. The reasoning in this case is similar to the previous case.

iii. If $\lambda = t \in \mathbb{R}$, a time passage, then
$\forall t' \in [0, t](v + t') \models I(\hat{Q}_{11}, D_{Q1}) \wedge \bigwedge_{k \in K} I(\eta_k)$, whence
$(v + t') \models I(\hat{Q}_{11}; \hat{Q}_{12}, D_{Q1})$.
Also $\hat{Q}_{11} = \hat{Q}'_{11}, D_{Q1} = D'_{Q1}, \eta = \eta', v' = v + t$. So
$((\hat{Q}_{11}; \hat{Q}_{12}, D_{Q1}), \eta, v) \xrightarrow{t} ((\hat{Q}_{11}; \hat{Q}_{12}, D_{Q1}), \eta, v + t)$.
By induction hypothesis, $\hat{Q}'_{11} \simeq \hat{Q}''_{11}$, $Q'_{11} \simeq Q''_{11}$, and
$((\hat{Q}''_{11}, D_{Q1}), \eta, v + t) \overset{(5.5)}{\longmapsto} (Q''_{11}, N', D')$ whence
$((\hat{Q}''_{11}; \hat{Q}_{12}, D'_{Q1}), \eta', v') \overset{(5.5)}{\longmapsto} (Q''_{11}; \texttt{unclk}(\hat{Q}_{12}), N', D')$ as required.

(b) A transition inferred by rule §4.3(Seq 2),

$(\mathtt{age}(\hat{Q}_{11}, v); \mathtt{unclk}(\hat{Q}_{12}), \mathtt{age}(\eta, v), D_{Q1}) \xrightarrow{\lambda} (\mathtt{unclk}(\hat{Q}_{12}), N', D')$
where
$\lambda = \lambda_p \in A_p$ and $(\mathtt{age}(\hat{Q}_{11}, v), \mathtt{age}(\eta, v), D_{Q1}) \xrightarrow{\lambda_p} (\surd, N', D')$.
By induction hypothesis there is a timed transition in the automaton formalism, $((\hat{Q}_{11}, D_{Q1}), \eta, v) \xrightarrow{\lambda_p} ((\hat{Q}'_{11}, D'_{Q1}), \eta', v')$ with
$(\exists \hat{Q}''_{11})\hat{Q}'_{11} \simeq \hat{Q}''_{11}$, $(\exists Q''_{11})Q'_{11} \simeq Q''_{11}$ and
$((\hat{Q}''_{11}, D'_{Q1}), \eta', v') \longmapsto (Q''_{11}, N', D')$ by mapping/relation (5.5).
From the definition of the age function and lemma B.1.8, $\hat{Q}''_{11} \equiv \surd, \hat{Q}'_{11} \equiv \surd$ and $Q''_{11} \equiv \surd$.
Now, $((\hat{Q}_{11}, D_{Q1}), \eta, v) \xrightarrow{\lambda_p} ((\surd, D'_{Q1}), \eta', v')$ must arise from a timed-automaton edge $(\hat{Q}_{11}, D_{Q1}) \xrightarrow{\zeta, \lambda_p, H} (\surd, D'_{Q1})$ where $v \models \zeta$ and $v' = v[H := 0] \models I(\surd, D'_{Q1})$. [There may be an additional synchronisation with an edge $\eta_k \longrightarrow \eta'_k$ in which case additional clock(s) are reset and $v' \models \eta'_k$ too. Otherwise $\eta'_k = \eta_k$.]
By §5.4.2.2(C_Seq 2) there is a timed-automaton edge $(\hat{Q}_{11}; \hat{Q}_{12}, D_{Q1}) \xrightarrow{\zeta, \lambda_p, H \cup H_{\hat{Q}_{12}}} (\hat{Q}_{12}, D'_{Q1})$ (where $H_{\hat{Q}_{12}}$ is the set of initial clock variables of $\hat{Q}_{12}$) which may also synchronise with an edge of a network channel automaton. By lemmas B.1.2 and B.1.4,
$v'' = v[H \cup H_{\hat{Q}_{12}} := 0] \models I(\hat{Q}_{12}, D'_{Q1}) \wedge I(\eta'_k)$
Hence, $((\hat{Q}_{11}; \hat{Q}_{12}, D_{Q1}), \eta, v) \xrightarrow{\lambda_p} ((\hat{Q}_{12}, D'_{Q1}), \eta', v'')$.
Since $((\surd, D'_{Q1}), \eta', v') \overset{(5.5)}{\longmapsto} (\surd, N', D'_{Q1})$ by induction hypothesis, $D' = D'_{D1}$ and by lemma B.1.5, $N' = \mathtt{age}(\eta', v'')$. Define $Q = \mathtt{age}(\hat{Q}, v'')$. Then $((\hat{Q}_{12}, D'_{Q1}), \eta', v'') \overset{(5.5)}{\longmapsto} (Q, N', D')$. By lemma B.1.6, $\mathtt{unclk}(\hat{Q}_{12}) \simeq Q$.

6. $(\hat{Q}_i, D_{Qi})_{i=1\ldots m}$ is a single term $(\hat{Q}_1, D_{Q1})$ where $\hat{Q}_1 = \hat{Q}_{11} + \hat{Q}_{12}$. This case is similar to the following, which is worked through in more detail.

7. $(\hat{Q}_i, D_{Qi})_{i=1\ldots m}$ is a single term $(\hat{Q}_1, D_{Q1})$ where $\hat{Q}_1 = \hat{Q}_{11}[> \hat{Q}_{12}$. In this case, let $Q_{1j} = \mathtt{age}(\hat{Q}_{1j}, v), j = 1, 2$. Then $\mathtt{age}(\hat{Q}_{11}[> \hat{Q}_{12}, v) = Q_{11}[> Q_{12}$ and we have
$((\hat{Q}_{11}[> \hat{Q}_{12}, D_{Q1}), \eta, v) \overset{(5.5)}{\longmapsto} (Q_{11}[> Q_{12}, N, D_{Q1}))$ where $N = \mathtt{age}(\eta, v)$.

There are now four sub-cases to consider.

(a) $(Q_{11}[> Q_{12}, N, D_{Q1}) \xrightarrow{\lambda_p \in A_p} (Q'_{11}[> Q_{12}, N', D'_{Q1})$ inferred by rule §4.3(Int 1), where $(Q_{11}, N, D_{Q1}) \xrightarrow{\lambda_p \in A_p} (Q'_{11}, N', D'_{Q1})$ and Q'_{11} is not $\surd$.

By induction hypothesis, $((\hat{Q}_{11}, D_{Q1}), \eta, v) \xrightarrow{\lambda_p} ((\hat{Q}'_{11}, \hat{D}'_{Q1}), \eta', v)$ and $\exists \hat{Q}''_{11} \simeq \hat{Q}'_{11}, Q''_{11} \simeq Q'_{11}$ such that: $((\hat{Q}''_{11}, D'_{Q1}), \eta', v') \overset{(5.5)}{\longmapsto} (Q''_{11}, N', D'_{Q1})$. This timed transition λ_p arises from an timed-automaton edge $(\hat{Q}_{11}, D_{Q1}) \xrightarrow{\zeta, \lambda_p, H} (\hat{Q}'_{11}, \hat{D}'_{Q1})$ (synchronising with an edge $\eta_k \longrightarrow \eta'_k$ if some $\eta'_k \neq \eta_k$) with $\neg \hat{Q}'_{11} \equiv \surd$, $v \models \zeta$ and $v' = v[H := 0] \models I(\hat{Q}'_{11}, \hat{D}'_{Q1})$. Also $v' \models$ the invariant of η'_k if appropriate; in this case a network clock is additionally reset.

So, by §5.4.2.2 (C_Int 1), there is a timed-automaton edge $(\hat{Q}_{11}[> \hat{Q}_{12}, D_{Q1}) \xrightarrow{\zeta, \lambda_p, H} (\hat{Q}'_{11}[> \hat{Q}_{12}, \hat{D}'_{Q1})$ possibly synchronising with a channel automaton edge.

Now $v[H := 0] \models I(\hat{Q}'_{11}, \hat{D}'_{Q1})\{\wedge I(\eta'_k)\}$ and $v \models I(\hat{Q}_{12}, D_{Q1})$. Since h^u is reset at every edge we can appeal to lemmas B.1.2, B.1.3 and infer $v[H := 0] \models I(\hat{Q}_{12}, \hat{D}'_{Q1})$ whence $v[H := 0] \models I(\hat{Q}'_{11}[> \hat{Q}_{12}, \hat{D}'_{Q1})\{\wedge I(\eta'_k)\}$. Since also $v \models \zeta$ there is a timed transition $((\hat{Q}_{11}[> \hat{Q}_{12}, D_{Q1}), \eta, v) \xrightarrow{\lambda_p} ((\hat{Q}'_{11}[> \hat{Q}_{12}, \hat{D}'_{Q1}), \eta', v)$ (H also includes a network channel clock to be reset, if necessary.)

Now, $\hat{Q}''_{11} \simeq \hat{Q}'_{11}$ and $Q''_{11} \simeq Q'_{11}$ so $\hat{Q}''_{11}[> \hat{Q}_{12} \simeq \hat{Q}'_{11}[> \hat{Q}_{12}$ and $Q''_{11}[> Q_{12} \simeq Q'_{11}[> Q_{12}$. Also $Q''_{11} = \mathsf{age}(\hat{Q}''_{11}, v[H := 0])$ by construction of the $\overset{(5.5)}{\longmapsto}$ relation, and $H \cap H_{\hat{Q}_{12}} = \emptyset$ assuming clocks are "safely allocated" so by lemma B.1.5, $Q = \mathsf{age}(\hat{Q}_{12}, v[H := 0])$ and $N' = \mathsf{age}(\eta', v[H := 0])$.

Then, by construction of the mapping relation, $((\hat{Q}''_{11}[> \hat{Q}_{12}, D'_{Q1}), \eta', v[H := 0]) \overset{(5.5)}{\longmapsto} (Q''_{11}[> Q_{12}, N', D'_{Q1})$.

(b) $(Q_{11}[> Q_{12}, N, D_{Q1}) \xrightarrow{\lambda_p \in A_p} (\surd, N', D'_{Q1})$ by rule §4.3(Int 2), where $(Q_{11}, N, D_{Q1}) \xrightarrow{\lambda_p} (\surd, N', D'_{Q1})$. This case is handled similarly.

(c) $(Q_{11}[> Q_{12}, N, D_{Q1}) \xrightarrow{\lambda_p \in A_p} (Q'_{12}, N', D'_{Q1})$ by rule §4.3(Int 3), where

$(Q_{12}, N, D_{Q1}) \xrightarrow{\lambda_p} (Q'_{12}, N', D'_{Q1})$. This case is handled similarly.

(d) $(Q_{11}[> Q_{12}, N, D_{Q1}) \xrightarrow{\lambda_{nt} \in A_n \cup \mathbb{R}} (Q'_{11}[> Q'_{12}, N', D'_{Q1})$ by §4.3(Int 4), where $(Q_{1j}, N, D_{Q1}) \xrightarrow{\lambda_{nt}} (Q'_{1j}, N', D'_{Q1})$ for $j = 1, 2$. There are two sub-sub-cases.

i. $\lambda_{nt} = \lambda_n \in A_n$. By lemma B.1.1, $Q_{11}[> Q_{12} \equiv Q'_{11}[> Q'_{12}$ and $D_{Q1} = D'_{Q1}$. By induction hypothesis, $((\hat{Q}_{1j}, D_{Q1}), \eta, v) \xrightarrow{\lambda_n} ((\hat{Q}_{1j}, D_{Q1}), \eta', v')$ for $j = 1, 2$; and $\exists \hat{Q}''_{11} \simeq \hat{Q}'_{11},\ \hat{Q}''_{12} \simeq \hat{Q}'_{12},\ Q''_{11} \simeq Q'_{11},\ Q''_{12} \simeq Q'_{12}$ such that $((\hat{Q}_{11}, D_{Q1}), \eta, v) \overset{(5.5)}{\longmapsto} (Q''_{11}, N', D_{Q1})$.
But $((\hat{Q}_{11}, D_{Q1}), \eta, v) \xrightarrow{\lambda_n} ((\hat{Q}_{11}, D_{Q1}), \eta', v')$ must arise from some edge $\eta_k \xrightarrow{\zeta, \lambda_n, H} \eta'_k$. The same edge yields $((\hat{Q}_{11}[> \hat{Q}_{12}, D_{Q1}), \eta, v) \xrightarrow{\lambda_n} ((\hat{Q}_{11}[> \hat{Q}_{12}, D_{Q1}), \eta', v')$.

ii. $\lambda_{nt} = t \in \mathbb{R}$. In this case $(\forall k)\eta_k = \eta'_k$ and $v' = v + t$. Also, $(\forall t' \in [0, t])v + t'$ satisfies all the location invariants of the η_k. The rest of the details are straightforward.

8. $(\hat{Q}_i, D_{Qi})_{i=1\ldots m}$ is a single term $(\hat{Q}_1, D_{Q1})$ where $\hat{Q}_1 = recX.\hat{Q}$. So $((recX.\hat{Q}, D_{Q1}), \eta, v) \overset{(5.5)}{\longmapsto} (\mathtt{age}(recX.\hat{Q}, v), \mathtt{age}(\eta, v), D_{Q1})$.
Now, $\mathtt{age}(recX.\hat{Q}, v) = \mathtt{age}(\hat{Q}[recX.\hat{Q}/X], v)$. The expression on the right hand side is "lower" in the recursive definition of age then the left, so whenever there is a transition
$(\mathtt{age}(\hat{Q}[recX.\hat{Q}/X], v), \mathtt{age}(\eta, v), D_{Q1}) \xrightarrow{\lambda} (Q'_1, N', D'_{Q1})$
we can infer by induction hypothesis
$\exists \hat{Q}''_1 \simeq \hat{Q}'_1,\ Q''_1 \simeq Q'_1$ and a corresponding timed transition
$((\hat{Q}[recX.\hat{Q}/X], D_{Q1}), \eta, v) \xrightarrow{\lambda} ((\hat{Q}''_1, D'_{Q1}), \eta', v')$ with
$((\hat{Q}''_1, D'_{Q1}), \eta', v') \overset{(5.5)}{\longmapsto} (Q''_1, N', D'_{Q1})$ (In particular, $\mathtt{age}(\eta', v') = N'$).

Two possible cases for λ need to be considered.

(a) A discrete action: $\lambda \in A_p \cup A_n$. This is a discrete process action or possibly and send or receive action synchronising with an edge of a channel automaton. A discrete transition $((\hat{Q}[recX.\hat{Q}/X], D_{Q1}), \eta, v) \xrightarrow{\lambda} ((\hat{Q}''_1, D'_{Q1}), \eta', v')$ must arise from

an automaton edge $(\hat{Q}[recX.\hat{Q}/X], D_{Q1}) \stackrel{\zeta,\lambda,H}{\longrightarrow} (\hat{Q}_1'', D_{Q1}')$ with $v \models \zeta$ and $v' \models I(\hat{Q}_1'', D_{Q1}')$, possibly synchronising with a channel automaton edge. From §4.3 rule (Rec), there is an edge $(recX.\hat{Q}, D_{Q1}) \stackrel{\zeta,\lambda,H}{\longrightarrow} (\hat{Q}_1'', D_{Q1}')$, and hence the required timed transition.

(b) A time passage: $\lambda = t \in \mathbb{R}$. Thus $((\hat{Q}[recX.\hat{Q}/X], D_{Q1}), \eta, v) \stackrel{t}{\longrightarrow} ((\hat{Q}_1'', D_{Q1}'), \eta', v')$ which must arise from passage of time at a (vector of) automaton locations: $v' = v + t$, $\hat{Q}_1'' = \hat{Q}[recX.\hat{Q}/X]$, $D_{Q1}' = D_{Q1}$, $\eta' = \eta$ and $\forall t' \in [0,t](v + t') \models I(recX.\hat{Q}, D_{Q1})$.

Hence there is a timed transition $((recX.\hat{Q}, D_{Q1}), \eta, v) \stackrel{t}{\longrightarrow} ((\hat{Q}_1'', D_{Q1}'), \eta', v')$ as required to complete the induction step.

9. $(\hat{Q}_i, D_{Qi})_{i=1\ldots m}$ is a vector (finite sequence) of clocked process terms. Without loss of generality we consider just a pair $(\hat{Q}_1, D_{Q1}), (\hat{Q}_2, D_{Q2})$ (this part of the proof generalises to m terms in a natural way). The mapping relation (5.5) takes the form: $((\hat{Q}_1, D_{Q1}), (\hat{Q}_2, D_{Q2}), \eta, v) \longmapsto (Q_1|Q_2, N, D_{Q1} \sqcup D_{Q2})$ where $Q_j = \mathsf{age}(\hat{Q}_j, v)$ $[j = 1, 2]$ and $N = \mathsf{age}(\eta) = (\mathsf{age}(\eta_k))_{k \in K}$. There are five sub-cases.

(a) $(Q_1|Q_2, N, D_{Q1} \sqcup D_{Q2}) \stackrel{\lambda_p \in A_p}{\longrightarrow} (Q_1'|Q_2, N', D_{Q1}' \sqcup D_{Q2})$ by §4.3 rule (Par 1), where $(Q_1, N, D_{Q1}) \stackrel{\lambda_p}{\longrightarrow} (Q_1', N', D_{Q1}')$ and $\neg Q_1' \equiv \surd$.

By induction hypothesis, $((\hat{Q}_1, D_{Q1})(\hat{Q}_2, D_{Q2}), \eta, v) \stackrel{\lambda_p}{\longrightarrow} ((\hat{Q}_1', D_{Q1}')(\hat{Q}_2, D_{Q2}), \eta', v')$ and $\exists \hat{Q}_1'' \simeq \hat{Q}_1'$, $Q_1'' \simeq Q_1'$ such that $((\hat{Q}_1'', D_{Q1}')(\hat{Q}_2, D_{Q2}), \eta', v') \stackrel{\lambda_p}{\longmapsto} (Q_1''|Q_2, N', D_{Q1}' \sqcup D_{Q2})$.
To complete this case, observe that $Q_1''|Q_2 \simeq Q_1'|Q_2$.

(b) $(Q_1|Q_2, N, D_{Q1} \sqcup D_{Q2}) \stackrel{\lambda_p \in A_p}{\longrightarrow} (Q_2, N', D_{Q1}' \sqcup D_{Q2})$ by §4.3 rule (Par 2), where $(Q_1, N, D_{Q1}) \stackrel{\lambda_p}{\longrightarrow} (\surd, N', D_{Q1}')$.

By induction hypothesis, $((\hat{Q}_1, D_{Q1})(\hat{Q}_2, D_{Q2}), \eta, v) \stackrel{\lambda_p}{\longrightarrow}$ $((\surd, D_{Q1}')(\hat{Q}_2, D_{Q2}), \eta', v') \simeq ((\hat{Q}_2, D_{Q1}' \sqcup D_{Q2}), \eta', v')$ with $N' = \mathsf{age}(\eta', v')$ and $\exists \hat{Q}_1'' \simeq \hat{Q}_1'$ such that

$$((\hat{Q}_1'', D_{Q1}')(\hat{Q}_2, D_{Q2}), \eta', v') \stackrel{\lambda_p}{\longmapsto} (Q_1''|Q_2, N', D_{Q1}' \sqcup D_{Q2}).$$

(c) $(Q_1|Q_2, N, D_{Q1} \sqcup D_{Q2}) \stackrel{\lambda_p \in A_p}{\longrightarrow} (Q_1|Q_2', N', D_{Q1} \sqcup D_{Q2}')$ by §4.3 rule (Par 3), where $(Q_2, N, D_{Q2}) \stackrel{\lambda_p}{\longrightarrow} (Q_2', N', D_{Q2}')$ and $\neg Q_2' \equiv \surd$. This is symmetrical to the (Par 1) case.

(d) $(Q_1|Q_2, N, D_{Q1} \sqcup D_{Q2}) \stackrel{\lambda_p \in A_p}{\longrightarrow} (Q_1, N', D_{Q1}' \sqcup D_{Q2})$ by §4.3 rule (Par 4), where $(Q_2, N, D_{Q2}) \stackrel{\lambda_p}{\longrightarrow} (\surd, N', D_{Q2}')$. This is symmetrical to the (Par 2) case.

(e) $(Q_1|Q_2, N, D_{Q1} \sqcup D_{Q2}) \stackrel{\lambda_{nt} \in A_n \cup \mathbb{R}}{\longrightarrow} (Q_1'|Q_2', N', D_{Q1} \sqcup D_{Q2})$ by §4.3 rule (Par 5), where $(Q_j, N, D_{Q1} \sqcup D_{Q2}) \stackrel{\lambda_{nt}}{\longrightarrow} (Q_j', N', D_{Q1} \sqcup D_{Q2})$, $j = 1, 2$. Let $D = D_{Q1} \sqcup D_{Q2}$ and recall $Q_j = \mathtt{age}(\hat{Q}_j, v)$ and $N = \mathtt{age}(\eta, v)$. By induction hypothesis,
$((\hat{Q}_j, D), \eta, v) \stackrel{\lambda_{nt}}{\longrightarrow} ((\hat{Q}_j', D), \eta', v')$ and
$\exists \hat{Q}_1'' \simeq \hat{Q}_1',\ Q_1'' \simeq Q_1',\ \hat{Q}_2'' \simeq \hat{Q}_2',\ Q_2'' \simeq Q_2'$, such that
$((\hat{Q}_j'', D), \eta', v) \stackrel{(5.5)}{\longmapsto} (Q_j'', N', D)$. The rest of the details are similar to the cases involving network actions and/or time passage.

Remark – these cases generalise to a vector of parallel process terms $(\hat{Q}_i, D_{Qi})_{i=1\ldots m}$ in a natural way.

B.4 A Transition in the TA Formalism is Simulated in bCANdle

The last part of the bisimulation proof is to show the reverse of the above: that given a pair of related states,
$((\hat{Q}_i, D_{Qi})_{i=1\ldots n}, (\eta_k)_{k\in K}, v) \stackrel{(5.5)}{\longmapsto} (|_{i=1}^{n} \mathtt{age}(\hat{Q}_i, v), (\mathtt{age}(\eta_k, v))_{k\in K}, \sqcup_{i=1}^{n} D_{Qi})$, for every transition in the timed-automaton formalism,
$((\hat{Q}_i, D_{Qi})_{i=1\ldots m}, (\eta_k)_{k\in K}, v) \stackrel{\lambda}{\longrightarrow} ((\hat{Q}_i', D_{Qi}')_{i=1\ldots m'}, (\eta_k')_{k\in K}, v')$ there is a transition with the same label λ from $(|_{i=1}^{m} \mathtt{age}(\hat{Q}_i, v), (\mathtt{age}(\eta_k, v))_{k\in K}, \sqcup_{i=1}^{m} D_{Qi})$ in the bCANdle timed transition system, and the λ-targets are strongly (semantically) equivalent to states related by the mapping.

The proof of this similar, and symmetrical, to that given in the previous section.

Again, this will be done by proving by induction on the recursive definition of the age function a slightly stronger property: for every pair of timed states
$((\hat{Q}_i, D_{Qi})_{i=1...m}, (\eta_k)_{k\in K}, v) \stackrel{(5.5)}{\longmapsto} (|_{i=1}^{m}\mathsf{age}(\hat{Q}_i, v), (\mathsf{age}(\eta_k, v))_{k\in K}, \sqcup_{i=1}^{n} D_{Qi})$ related by the mapping, for every transition
$((\hat{Q}_i, D_{Qi})_{i=1...m}, (\eta_k)_{k\in K}, v) \xrightarrow{\lambda} ((\hat{Q}'_i, D'_{Qi})_{i=1...m'}, (\eta'_k)_{k\in K}, v')$ in the timed-automaton formalism there is a transition with the same label
$(|_{i=1}^{m}\mathsf{age}(\hat{Q}_i, v), (\mathsf{age}(\eta_k, v))_{k\in K}, \sqcup_{i=1}^{m} D_{Qi}) \xrightarrow{\lambda} (Q', N', D')$ in the bCANdle timed transition system , where for some clocked bCANdle terms, $\hat{Q}''_i \simeq \hat{Q}'_i, (i = 1...m')$,
$((\hat{Q}''_i, D'_{Qi})_{i=1...m'}, (\eta'_k)_{k\in K}, v) \stackrel{(5.5)}{\longmapsto} (|_{i=1}^{m'}\mathsf{age}(\hat{Q}''_i, v), (\mathsf{age}(\eta'_k, v'))_{k\in K}, \sqcup_{i=1}^{m'} D'_{Qi})$ and $|_{i=1}^{m'}\mathsf{age}(\hat{Q}''_i, v') \simeq Q'$ and $N' = \mathsf{age}(\eta', v')$ and $D' = \sqcup_{i=1}^{m'} D'_{Qi}$.

Proof There are various cases to be considered: time passage with discrete action, discrete action internal to a channel automaton, and discrete action on a process term (several cases depending on $(\hat{Q}_i, D_{Qi})_{i=1...m}$). As before, a vector of channel automaton locations is abbreviated by suppressing the subscripts: $(\eta_k)_{k\in K}$ is abbreviated η, $(\mathsf{age}(\eta_k, v))_{k\in K}$ is abbreviated $\mathsf{age}(\eta, v)$ and so forth.

1. $\lambda = t \in \mathbb{R}$. Then $((\hat{Q}_i, D_{Qi})_{i=1...m}, \eta, v) = ((\hat{Q}'_i, D'_{Qi})_{i=1...m'}, \eta', v')$ – $m' = m$, $v' = v + t$, and $\forall k.\eta'_k = \eta_k$ and $\forall t' \in [0, t].v + t' \models I(\eta_k)$, and $\forall i.\hat{Q}'_i = \hat{Q}_i,\ D'_{Qi} = D_{Qi}$ and $\forall t' \in [0, t].v + t' \models I(\hat{Q}_i, D_{Qi})$.

   From this one can deduce a bCANdle network transition $\mathsf{age}(\eta, v) \xrightarrow{t} \mathsf{age}(\eta, v + t)$ by employing lemma B.1.7. To further infer $(|_{i=1}^{m}\mathsf{age}(\hat{Q}_i, v), \mathsf{age}(\eta, v), \sqcup_{i=1}^{n} D_{Qi}) \xrightarrow{t} (|_{i=1}^{m}\mathsf{age}(\hat{Q}_i, v + t), \mathsf{age}(\eta, v + t), \sqcup_{i=1}^{n} D_{Qi})$, as required, we make use of §4.3 rules (Snd 3), (Rcv 3), (Comp 3), (Gu 3), (Ch 3), (Int 4), (Par 5).

2. $\lambda = \lambda_n$, an internal discrete action in just one of the channel automata, say, $\eta_k \longrightarrow \eta'_k$. Then $v = v'$, $m = m'$, $\forall i.\hat{Q}_i = \hat{Q}'_i$, $D_{Qi} = D'_{Qi}$, and $\forall l \neq k.\eta'_l = \eta_l$. By construction of the age function, there is a network

transition $\mathsf{age}(\eta_k, v) \xrightarrow{\lambda_n} \mathsf{age}(\eta'_k, v)$, from which a timed transition $(|_{i=1}^m \mathsf{age}(\hat{Q}_i, v), \mathsf{age}(\eta, v), \sqcup_{i=1}^n D_{Qi}) \xrightarrow{t} (|_{i=1}^m \mathsf{age}(\hat{Q}_i, v'), \mathsf{age}(\eta, v'), \sqcup_{i=1}^n D_{Qi})$ may be inferred using §4.3 rules (Snd 2), (Rcv 2), (Comp 2), (Gu 2), (Ch 3), (Int 4), (Par 5).

In the remaining cases actions λ arising as time passage or as network actions are handled much as above; the cases below are those of discrete actions on "process term" components arising via the rules of §5.4.2.2. In these cases $\lambda \in A_p$.

3. $(\hat{Q}_i, D_{Qi})_{i=1...m}$ is a single term $(k!i.x, D_{Q1})$ where $k \in K, i \in I, x \in Var$. λ must be $((\hat{Q}_1, D_{Q1}), \eta, v) \xrightarrow{k!i.v} ((\surd, D_{Q1}), \eta', v[h^u := 0])$ arising by rule §5.4.2.2(C_Snd), where $\hat{Q}_1 = k!i.x$, $v = D_{Q1}.x$. A synchronous action $\eta_k \longrightarrow \eta'_k$ in channel k adds message $i.v$ to its queue; for $l \neq k$, $\eta_l = \eta'_l$.

   By rule §4.3(Snd 1) there is a bCANdle transition $(k!i.x, N, D_{Q1}) \xrightarrow{k!i.v} (\surd, N[k := (s, u \leftarrow\!\!\!\rho\ i.v)], D_{Q1})$ where $\mathsf{age}(k!i.x, v) = k!i.x$ and $N = \mathsf{age}(\eta, v)$, $N[k := (s, u \leftarrow\!\!\!\rho\ i.v)] = \mathsf{age}(\eta', v)$.

4. $(\hat{Q}_i, D_{Qi})_{i=1...m}$ is a single term $(k?i.x, D_{Q1})$ where $k \in K, i \in I, x \in Var$. λ must be $((\hat{Q}_1, D_{Q1}), \eta, v) \xrightarrow{k?i.v} ((\surd, D'_{Q1}), \eta', v[h^u := 0])$ arising by rule §5.4.2.2(C_Rcv), where $\hat{Q}_1 = k?i.x$, $D'_{Q1} = D_{Q1}[x := v]$. $\eta'_l = \eta_l for l \neq k$ and there is a transition $\eta_k \longrightarrow \eta'_k$ from the acceptance point of channel k broadcasting a synchronisation to communicate the data value.

   By rule §4.3(Rcv 1) there is then a bCANdle transition $(k?i.x, N, D_{Q1}) \xrightarrow{k!i.v} (\surd, N', D'_{Q1})$ where $\mathsf{age}(k?i.x, v) = k?i.x$ and $N = \mathsf{age}(\eta, v)$, $N' = \mathsf{age}(\eta', v)$, $D'_{Q1} = D_{Q1}[x := v]$.

5. $(\hat{Q}_i, D_{Qi})_{i=1...m}$ is a single term $([\omega : t_1, t_2]^h, D_{Q1})$. A transition from $(([\omega : t_1, t_2]^h, D_{Q1}), \eta, v)$ must incorporate an edge from $([\omega : t_1, t_2]^h, D_{Q1})$ and this can only arise from rule §5.4.2.2(C_Cmp). It follows that λ is $(([\omega : t_1, t_2]^h, D_{Q1}), \eta, v) \xrightarrow{\omega} ((\surd, D'_{Q1}), \eta, v')$ where $D_{Q1} \xrightarrow{\omega}_d D'_{Q1}$, that is, D'_{Q1} is the state of D_{Q1} after the computation ω has run, and $v \models h \geq t_1$, $v' = v[h^u := 0]$.

$\mathtt{age}([\omega : t_1, t_2]^h) = [\omega : t_1 \dot{-} v(h), t_2 \dot{-} v(h)] = [\omega : 0, t_2 \dot{-} v(h)]$ and so the corresponding bCANdle transition takes the form $([\omega : t_1 \dot{-} v(h), t_2 \dot{-} v(h)], \mathtt{age}(\eta, v), D_{Q1}) \xrightarrow{\omega} (\surd, \eta, D'_{Q1})$ which exists by virtue of §4.3 rule (Comp 1).

6. $(\hat{Q}_i, D_{Qi})_{i=1\ldots m}$ is a single term $(\hat{Q}_1, D_{Q1})$ where $\hat{Q}_1 \equiv \gamma \to \hat{Q}$. Recall that $\mathtt{age}(\gamma \to \hat{Q}) = \gamma \to \mathtt{unclk}(\hat{Q})$. The timed transition $((\gamma \to \hat{Q}, D_{Q1}), \eta, v) \xrightarrow{\lambda} ((\hat{Q}'_1, D'_{Q1}), \eta', v')$ must arise via §5.4.2.2(C_Gd) from an edge $(\gamma \to \hat{Q}, D_{Q1}) \overset{1,\gamma,\{h^u\}\cup H_{\hat{Q}}}{\longrightarrow} (\hat{Q}, D_{Q1})$ where $D_{Q1} \models \gamma$. Thus, $\lambda = \gamma$, $v' = v[\{h^u\} \cup H_{\hat{Q}} := 0]$, $\eta' = \eta$, $D'_{Q1} = D_{Q1}$ and $\hat{Q}'_1 = \hat{Q}$.
By §4.3 rule (Gu 1) there is a bCANdle transition $(\gamma \to \mathtt{unclk}(\hat{Q}), \mathtt{age}(\eta, v), D_{Q1}) \xrightarrow{\gamma} (\mathtt{unclk}(\hat{Q}), \mathtt{age}(\eta, v), D_{Q1})$ which provides the required corresponding bCANdle transition.

7. $(\hat{Q}_i, D_{Qi})_{i=1\ldots m}$ is a single term $(\hat{Q}_{11}; \hat{Q}_{12}, D_{Q1})$. A timed transition λ from $((\hat{Q}_{11}; \hat{Q}_{12}, D_{Q1}), \eta, v)$ can arise in one of two ways.

   (a) $((\hat{Q}_{11}; \hat{Q}_{12}, D_{Q1}), \eta, v) \xrightarrow{\lambda} ((\hat{Q}'_{11}; \hat{Q}_{12}, D'_{Q1}), \eta, v')$ arising from an edge $(\hat{Q}_{11}; \hat{Q}_{12}, D_{Q1}) \xrightarrow{\zeta,\lambda,H} (\hat{Q}'_{11}; \hat{Q}_{12}, D'_{Q1})$ derived by rule (C_Seq 1) of §5.4.2.2 from an edge $(\hat{Q}_{11}, D_{Q1}) \xrightarrow{\zeta,\lambda,H} (\hat{Q}'_{11}, D'_{Q1})$ where $\hat{Q}'_{11}$ is not $\surd$.

   Thus, $v \models \zeta$ and $v' = v[H := 0] \models I(\hat{Q}'_{11}; \hat{Q}_{12})$; so, $v' \models I(\hat{Q}'_{11})$ and there is a timed transition $((\hat{Q}_{11}, D_{Q1}), \eta, v) \xrightarrow{\lambda} ((\hat{Q}'_{11}, D'_{Q1}), \eta, v')$.

   By induction hypothesis, there is a bCANdle transition corresponding to this under the bisimulation: $(\mathtt{age}(\hat{Q}_{11}, v), \mathtt{age}(\eta, v), D_{Q1}) \xrightarrow{\lambda} (Q'', \mathtt{age}(\eta, v), D'_{Q1})$ where for some $\hat{Q}''_{11} \simeq \hat{Q}'_{11}$, $((\hat{Q}''_{11}, D'_{Q1}), \eta, v') \overset{(5.5)}{\longmapsto} (\mathtt{age}(\hat{Q}''_{11}, v'), \mathtt{age}(\eta, v'), D'_{Q1})$ with $\mathtt{age}(\eta, v) = \mathtt{age}(\eta, v')$ and $Q'' \simeq \mathtt{age}(\hat{Q}''_{11}, v')$.

   §4.3 rule (Seq 1) implies a transition $(\mathtt{age}(\hat{Q}_{11}; \hat{Q}_{12}, v), \mathtt{age}(\eta, v), D_{Q1}) \xrightarrow{\lambda} (Q''; \mathtt{unclk}(\hat{Q}_{12}), \mathtt{age}(\eta, v), D'_{Q1})$ (remember $\mathtt{age}(\hat{Q}_{11}; \hat{Q}_{12}, v) = \mathtt{age}(\hat{Q}_{11}, v); \mathtt{unclk}(\hat{Q}_{12})$) from which the induction can be established.

(b) $((\hat{Q}_{11}; \hat{Q}_{12}, D_{Q1}), \eta, v) \xrightarrow{\lambda} ((\hat{Q}_{12}, D'_{Q1}), \eta, v')$ arising from an edge $(\hat{Q}_{11}; \hat{Q}_{12}, D_{Q1}) \xrightarrow{\zeta,\lambda,H\cup H_{\hat{Q}_{12}}} (\hat{Q}_{12}, D'_{Q1})$ derived by rule (C_Seq 2) of §5.4.2.2 from an edge $(\hat{Q}_{11}, D_{Q1}) \xrightarrow{\zeta,\lambda,H} (\surd, D'_{Q1})$.

Thus, $v \models \zeta$ and $v' = v[H \cup H_{\hat{Q}_{12}} := 0] \models I(\hat{Q}_{12})$; so there is a timed transition $((\hat{Q}_{11}, D_{Q1}), \eta, v) \xrightarrow{\lambda} ((\surd, D'_{Q1}), \eta, v')$. The rest of this part of the argument from the induction hypothesis is similar to the previous part, but using §4.3 rule (Seq 2) in lieu of (Seq 1).

8. $(\hat{Q}_i, D_{Qi})_{i=1\ldots m}$ is a single term $(\hat{Q}_{11}[> \hat{Q}_{12}, D_{Q1})$. A timed transition λ from $((\hat{Q}_{11}[> \hat{Q}_{12}, D_{Q1}), \eta, v)$ can arise in one of three ways.

(a) $((\hat{Q}_{11}[> \hat{Q}_{12}, D_{Q1}), \eta, v) \xrightarrow{\lambda} ((\hat{Q}'_{11}[> \hat{Q}_{12}, D'_{Q1}), \eta, v')$ arising from an edge $(\hat{Q}_{11}[> \hat{Q}_{12}, D_{Q1}) \xrightarrow{\zeta,\lambda,H} (\hat{Q}'_{11}[> \hat{Q}_{12}, D'_{Q1})$ derived by rule (C_Int 1) of §5.4.2.2 from an edge $(\hat{Q}_{11}, D_{Q1}) \xrightarrow{\zeta,\lambda,H} (\hat{Q}'_{11}, D'_{Q1})$ where $\hat{Q}'_{11}$ is not $\surd$.

Thus, $v \models \zeta$ and $v' = v[H := 0] \models I(\hat{Q}'_{11}[> \hat{Q}_{12})$; so, $v' \models I(\hat{Q}'_{11})$ and there is a timed transition $((\hat{Q}_{11}, D_{Q1}), \eta, v) \xrightarrow{\lambda} ((\hat{Q}'_{11}, D'_{Q1}), \eta, v')$.

By induction hypothesis, there is a bCANdle transition corresponding to this under the bisimulation: $(\mathsf{age}(\hat{Q}_{11}, v), \mathsf{age}(\eta, v), D_{Q1}) \xrightarrow{\lambda} (Q'', \mathsf{age}(\eta, v), D'_{Q1})$ where for some $\hat{Q}''_{11} \simeq \hat{Q}'_{11}$, $((\hat{Q}''_{11}, D'_{Q1}), \eta, v') \overset{(5.5)}{\longmapsto} (\mathsf{age}(\hat{Q}''_{11}, v'), \mathsf{age}(\eta, v'), D'_{Q1})$ with $\mathsf{age}(\eta, v) = \mathsf{age}(\eta, v')$ and $Q'' \simeq \mathsf{age}(\hat{Q}''_{11}, v')$.

§4.3 rule (Int 1) implies a transition $(\mathsf{age}(\hat{Q}_{11}[> \hat{Q}_{12}, v), \mathsf{age}(\eta, v), D_{Q1}) \xrightarrow{\lambda} (Q''[> \mathsf{age}(\hat{Q}_{12}, v), \mathsf{age}(\eta, v), D'_{Q1})$ (remember $\mathsf{age}(\hat{Q}_{11}[> \hat{Q}_{12}, v) = \mathsf{age}(\hat{Q}_{11}, v)[> \mathsf{age}(\hat{Q}_{12}, v))$ from which the induction can be established.

(b) $((\hat{Q}_{11}[> \hat{Q}_{12}, D_{Q1}), \eta, v) \xrightarrow{\lambda} ((\surd, D'_{Q1}), \eta, v')$ arising from an edge $(\hat{Q}_{11}[> \hat{Q}_{12}, D_{Q1}) \xrightarrow{\zeta,\lambda,H} (\surd, D'_{Q1})$ derived by rule (C_Int 2) of §5.4.2.2 from an edge $(\hat{Q}_{11}, D_{Q1}) \xrightarrow{\zeta,\lambda,H} (\surd, D'_{Q1})$.

Thus, $v \models \zeta$ and $v' = v[H := 0]$; there is a timed transition

$((\hat{Q}_{11}, D_{Q1}), \eta, v) \xrightarrow{\lambda} ((\surd, D'_{Q1}), \eta, v')$. The rest of this part of the argument from induction hypothesis is similar to the previous part, but using §4.3 rule (Int 2) in lieu of (Int 1).

(c) $((\hat{Q}_{11} [> \hat{Q}_{12}, D_{Q1}), \eta, v) \xrightarrow{\lambda} ((\hat{Q}'_{12}, D'_{Q1}), \eta, v')$ arising from an edge $(\hat{Q}_{11} [> \hat{Q}_{12}, D_{Q1}) \xrightarrow{\zeta,\lambda,H} (\hat{Q}'_{12}, D'_{Q1})$ derived by rule (C_Int 3) of §5.4.2.2 from an edge $(\hat{Q}_{12}, D_{Q1}) \xrightarrow{\zeta,\lambda,H} (\hat{Q}'_{12}, D'_{Q1})$.
Thus, $v \models \zeta$ and $v' = v[H := 0] \models I(\hat{Q}'_{12})$; so there is a timed transition $((\hat{Q}_{12}, D_{Q1}), \eta, v) \xrightarrow{\lambda} ((\hat{Q}'_{12}, D'_{Q1}), \eta, v')$. The rest of this part of the argument from induction hypothesis is similar to the previous part, using §4.3 rule (Int 3).

9. $(\hat{Q}_i, D_{Qi})_{i=1...m}$ is a single term $(\hat{Q}_{11} + \hat{Q}_{12}, D_{Q1})$. A timed transition λ from $((\hat{Q}_{11} + \hat{Q}_{12}, D_{Q1}), \eta, v)$ must arise either as

   (a) $((\hat{Q}_{11} + \hat{Q}_{12}, D_{Q1}), \eta, v) \xrightarrow{\lambda} ((\hat{Q}'_{11}, D'_{Q1}), \eta, v')$ arising from an edge $(\hat{Q}_{11} + \hat{Q}_{12}, D_{Q1}) \xrightarrow{\zeta,\lambda,H} (\hat{Q}'_{11}, D'_{Q1})$ derived by rule §5.4.2.2 (C_Ch 1) from an edge $(\hat{Q}_{11}, D_{Q1}) \xrightarrow{\zeta,\lambda,H} (\hat{Q}'_{11}, D'_{Q1})$. This is similar to the previous two groups of cases.

   (b) $((\hat{Q}_{11} + \hat{Q}_{12}, D_{Q1}), \eta, v) \xrightarrow{\lambda} ((\hat{Q}'_{12}, D'_{Q1}), \eta, v')$ arising from an edge $(\hat{Q}_{11} + \hat{Q}_{12}, D_{Q1}) \xrightarrow{\zeta,\lambda,H} (\hat{Q}'_{12}, D'_{Q1})$ derived by rule §5.4.2.2 (C_Ch 2) from an edge $(\hat{Q}_{12}, D_{Q1}) \xrightarrow{\zeta,\lambda,H} (\hat{Q}'_{12}, D'_{Q1})$. This is symmetrical to the previous case.

10. $(\hat{Q}_i, D_{Qi})_{i=1...m}$ is a single term $(recX.\hat{Q}, D_{Q1})$.
$((recX.\hat{Q}, D_{Q1}), \eta, v) \xrightarrow{\lambda} ((\hat{Q}', D'_{Q1}), \eta, v')$ must arise from an edge $(recX.\hat{Q}, D_{Q1}) \xrightarrow{\zeta,\lambda,H} (\hat{Q}', D'_{Q1})$ derived by rule §5.4.2.2 (C_Rec) from an edge $(\hat{Q}[recX.\hat{Q}/X], D_{Q1}) \xrightarrow{\zeta,\lambda,H} (\hat{Q}', D'_{Q1})$. We have $v \models \zeta$ and $v' = v[H := 0] \models I(\hat{Q}', D'_{Q1})$, and hence a timed transition $((\hat{Q}[recX.\hat{Q}/X], D_{Q1}), \eta, v) \xrightarrow{\lambda} ((\hat{Q}', D'_{Q1}), \eta, v')$.

   By induction hypothesis there is a bCANdle transition $(\mathsf{age}(\hat{Q}[recX.\hat{Q}/X], v), \mathsf{age}(\eta, v), D_{Q1}) \xrightarrow{\lambda} (Q'', \mathsf{age}(\eta, v), D'_{Q1})$ where for some $\hat{Q}'' \simeq \hat{Q}'$, $((\hat{Q}'', D'_{Q1}), \eta, v') \overset{(5.5)}{\longmapsto} (\mathsf{age}(\hat{Q}'', v'), \mathsf{age}(\eta, v'), D'_{Q1})$ with $\mathsf{age}(\eta, v) = \mathsf{age}(\eta, v')$ and $Q'' \simeq \mathsf{age}(\hat{Q}'', v')$.

§4.3 rule (Rec) implies a bCANdle transition
$(\mathsf{age}(recX.\hat{Q}, v), \mathsf{age}(\eta, v), D_{Q1}) \xrightarrow{\lambda} (Q'', \mathsf{age}(\eta, v), D'_{Q1})$ which, because $(\mathsf{age}(recX.\hat{Q}, v) \equiv (\mathsf{age}(\hat{Q}[recX.\hat{Q}/X], v)$, establishes the induction.

11. $(\hat{Q}_i, D_{Qi})_{i=1\ldots m}$ is a pair $(\hat{Q}_i, D_{Qi})_{i=1,2}$ of parallel process terms and $((\hat{Q}_i, D_{Qi})_{i=1,2}, \eta, v) \xrightarrow{\lambda} (\hat{Q}'_i, D'_{Qi})_{i=1,2}, \eta', v')$ is a discrete transition on one of the precess terms.

    (a) λ arises from an edge
    $(\hat{Q}_1, D_{Q1}) \xrightarrow{\zeta,\lambda,H} (\hat{Q}'_1, D'_{Q1})$ and $\hat{Q}'_1$ is not $\surd$ but $\hat{Q}'_2 \equiv \hat{Q}_2$. Then, $v \models \zeta$ and $v' = v[H := 0] \models I(\hat{Q}'_1, D'_{Q1}) \wedge I(\hat{Q}'_2, D'_{Q2}) \wedge \bigwedge_{k\in K} I(\eta_k)$. In this case the required $(\mathsf{age}(\hat{Q}_1, v) | \mathsf{age}(\hat{Q}_2, v), \mathsf{age}(\eta, v), D_{Q1} \sqcup D_{Q2}) \xrightarrow{\lambda} (\mathsf{age}(\hat{Q}'_1, v') | \mathsf{age}(\hat{Q}'_2, v'), \mathsf{age}(\eta', v'), D'_{Q1} \sqcup D'_{Q2})$ may be inferred from
    $(\mathsf{age}(\hat{Q}_1, v), \mathsf{age}(\eta, v), D_{Q1}) \xrightarrow{\lambda} (\mathsf{age}(\hat{Q}'_1, v'), \mathsf{age}(\eta', v'), D'_{Q1})$ using §4.3 rule (Par 1).

    (b) λ arises from an edge
    $(\hat{Q}_1, D_{Q1}) \xrightarrow{\zeta,\lambda,H} (\surd, D'_{Q1})$. The reasoning is similar to the previous case, but uses §4.3 rule (Par 2).

    (c) λ arises from an edge
    $(\hat{Q}_2, D_{Q2}) \xrightarrow{\zeta,\lambda,H} (\hat{Q}'_2, D'_{Q2})$. The two cases (whether or not $\hat{Q}'_2$ is $\surd$) are symmetrical with the previous two cases and employ §4.3 rules (Par 3), (Par 4).

    (d) Cases in which $(\hat{Q}_i, D_{Qi})_{i=1\ldots m}$ is *several* parallel process terms generalise on these cases in a natural way.

Appendix C

CANGen Generated Code for a CAN Channel

```
process Channel(const int l, const int u, const int L, const int U,
    urgent chan & nMsg_000_000, urgent chan & nMsg_000_001,
    urgent chan & nMsg_000_002, urgent chan & nMsg_000_003,
    urgent chan & nMsg_001_000, urgent chan & nMsg_001_001,
    urgent chan & nMsg_001_002, urgent chan & nMsg_001_003,
    urgent chan & nMsg_002_000, urgent chan & nMsg_002_001,
    urgent chan & nMsg_002_002, urgent chan & nMsg_002_003,
    broadcast chan & dMsg_000_000, broadcast chan & dMsg_000_001,
    broadcast chan & dMsg_000_002, broadcast chan & dMsg_000_003,
    broadcast chan & dMsg_001_000, broadcast chan & dMsg_001_001,
    broadcast chan & dMsg_001_002, broadcast chan & dMsg_001_003,
    broadcast chan & dMsg_002_000, broadcast chan & dMsg_002_001,
    broadcast chan & dMsg_002_002, broadcast chan & dMsg_002_003) {
  clock h;
  int[0,3] ip;
  int[0,3] vp;
  int[-1,3] Q[3];

state
  Init, Free,
  M2S_000_000, M2S_000_001, M2S_000_002, M2S_000_003, M2S_001_000,
  M2S_001_001, M2S_001_002, M2S_001_003, M2S_002_000, M2S_002_001,
  M2S_002_002, M2S_002_003,
```

```
  PreAcc_000_000, PreAcc_000_001, PreAcc_000_002, PreAcc_000_003,
  PreAcc_001_000, PreAcc_001_001, PreAcc_001_002, PreAcc_001_003,
  PreAcc_002_000, PreAcc_002_001, PreAcc_002_002, PreAcc_002_003,
  Acc_000_000, Acc_000_001, Acc_000_002, Acc_000_003, Acc_001_000,
  Acc_001_001, Acc_001_002, Acc_001_003, Acc_002_000, Acc_002_001,
  Acc_002_002, Acc_002_003,
  PostAcc_000_000, PostAcc_000_001, PostAcc_000_002, PostAcc_000_003,
  PostAcc_001_000, PostAcc_001_001, PostAcc_001_002, PostAcc_001_003,
  PostAcc_002_000, PostAcc_002_001, PostAcc_002_002, PostAcc_002_003,
  PreAcc{h<=u}, Acc, PostAcc{h<=U},
  PostChk_000, PostChk_001, PostChk_002;

  commit
    Init,
    M2S_000_000, M2S_000_001, M2S_000_002, M2S_000_003, M2S_001_000,
    M2S_001_001, M2S_001_002, M2S_001_003, M2S_002_000, M2S_002_001,
    M2S_002_002, M2S_002_003,
    PreAcc_000_000, PreAcc_000_001, PreAcc_000_002, PreAcc_000_003,
    PreAcc_001_000, PreAcc_001_001, PreAcc_001_002, PreAcc_001_003,
    PreAcc_002_000, PreAcc_002_001, PreAcc_002_002, PreAcc_002_003,
    Acc_000_000, Acc_000_001, Acc_000_002, Acc_000_003, Acc_001_000,
    Acc_001_001, Acc_001_002, Acc_001_003, Acc_002_000, Acc_002_001,
    Acc_002_002, Acc_002_003,
    PostAcc_000_000, PostAcc_000_001, PostAcc_000_002, PostAcc_000_003,
    PostAcc_001_000, PostAcc_001_001, PostAcc_001_002, PostAcc_001_003,
    PostAcc_002_000, PostAcc_002_001, PostAcc_002_002, PostAcc_002_003,
    PostChk_000, PostChk_001, PostChk_002;
  urgent
    Acc;
  init
    Init;

trans
  Init -> Init {
    guard ip < 3;
    assign Q[ip] = -1, ip++, h = 0;
  },
  Init -> Free { guard ip == 3; assign h = 0; },

  Free -> M2S_000_000 { sync nMsg_000_000?; },
  Free -> M2S_000_001 { sync nMsg_000_001?; },
```

```
Free -> M2S_000_002 { sync nMsg_000_002?; },
Free -> M2S_000_003 { sync nMsg_000_003?; },
Free -> M2S_001_000 { sync nMsg_001_000?; },
Free -> M2S_001_001 { sync nMsg_001_001?; },
Free -> M2S_001_002 { sync nMsg_001_002?; },
Free -> M2S_001_003 { sync nMsg_001_003?; },
Free -> M2S_002_000 { sync nMsg_002_000?; },
Free -> M2S_002_001 { sync nMsg_002_001?; },
Free -> M2S_002_002 { sync nMsg_002_002?; },
Free -> M2S_002_003 { sync nMsg_002_003?; },

M2S_000_000 -> PreAcc { assign ip=0, vp=0, h = 0; },
M2S_000_001 -> PreAcc { assign ip=0, vp=1, h = 0; },
M2S_000_002 -> PreAcc { assign ip=0, vp=2, h = 0; },
M2S_000_003 -> PreAcc { assign ip=0, vp=3, h = 0; },
M2S_001_000 -> PreAcc { assign ip=1, vp=0, h = 0; },
M2S_001_001 -> PreAcc { assign ip=1, vp=1, h = 0; },
M2S_001_002 -> PreAcc { assign ip=1, vp=2, h = 0; },
M2S_001_003 -> PreAcc { assign ip=1, vp=3, h = 0; },
M2S_002_000 -> PreAcc { assign ip=2, vp=0, h = 0; },
M2S_002_001 -> PreAcc { assign ip=2, vp=1, h = 0; },
M2S_002_002 -> PreAcc { assign ip=2, vp=2, h = 0; },
M2S_002_003 -> PreAcc { assign ip=2, vp=3, h = 0; },

PreAcc -> PreAcc_000_000 { sync nMsg_000_000?; },
PreAcc -> PreAcc_000_001 { sync nMsg_000_001?; },
PreAcc -> PreAcc_000_002 { sync nMsg_000_002?; },
PreAcc -> PreAcc_000_003 { sync nMsg_000_003?; },
PreAcc -> PreAcc_001_000 { sync nMsg_001_000?; },
PreAcc -> PreAcc_001_001 { sync nMsg_001_001?; },
PreAcc -> PreAcc_001_002 { sync nMsg_001_002?; },
PreAcc -> PreAcc_001_003 { sync nMsg_001_003?; },
PreAcc -> PreAcc_002_000 { sync nMsg_002_000?; },
PreAcc -> PreAcc_002_001 { sync nMsg_002_001?; },
PreAcc -> PreAcc_002_002 { sync nMsg_002_002?; },
PreAcc -> PreAcc_002_003 { sync nMsg_002_003?; },

PreAcc_000_000 -> PreAcc { assign Q[0] = 0; },
PreAcc_000_001 -> PreAcc { assign Q[0] = 1; },
PreAcc_000_002 -> PreAcc { assign Q[0] = 2; },
PreAcc_000_003 -> PreAcc { assign Q[0] = 3; },
```

```
PreAcc_001_000 -> PreAcc { assign Q[1] = 0; },
PreAcc_001_001 -> PreAcc { assign Q[1] = 1; },
PreAcc_001_002 -> PreAcc { assign Q[1] = 2; },
PreAcc_001_003 -> PreAcc { assign Q[1] = 3; },
PreAcc_002_000 -> PreAcc { assign Q[2] = 0; },
PreAcc_002_001 -> PreAcc { assign Q[2] = 1; },
PreAcc_002_002 -> PreAcc { assign Q[2] = 2; },
PreAcc_002_003 -> PreAcc { assign Q[2] = 3; },

PreAcc -> Acc { guard h >= l;  assign h = 0; },

Acc -> Acc_000_000 { guard ip == 0 && vp == 0; },
Acc -> Acc_000_001 { guard ip == 0 && vp == 1; },
Acc -> Acc_000_002 { guard ip == 0 && vp == 2; },
Acc -> Acc_000_003 { guard ip == 0 && vp == 3; },
Acc -> Acc_001_000 { guard ip == 1 && vp == 0; },
Acc -> Acc_001_001 { guard ip == 1 && vp == 1; },
Acc -> Acc_001_002 { guard ip == 1 && vp == 2; },
Acc -> Acc_001_003 { guard ip == 1 && vp == 3; },
Acc -> Acc_002_000 { guard ip == 2 && vp == 0; },
Acc -> Acc_002_001 { guard ip == 2 && vp == 1; },
Acc -> Acc_002_002 { guard ip == 2 && vp == 2; },
Acc -> Acc_002_003 { guard ip == 2 && vp == 3; },

Acc_000_000 -> PostAcc {sync dMsg_000_000!; },
Acc_000_001 -> PostAcc {sync dMsg_000_001!; },
Acc_000_002 -> PostAcc {sync dMsg_000_002!; },
Acc_000_003 -> PostAcc {sync dMsg_000_003!; },
Acc_001_000 -> PostAcc {sync dMsg_001_000!; },
Acc_001_001 -> PostAcc {sync dMsg_001_001!; },
Acc_001_002 -> PostAcc {sync dMsg_001_002!; },
Acc_001_003 -> PostAcc {sync dMsg_001_003!; },
Acc_002_000 -> PostAcc {sync dMsg_002_000!; },
Acc_002_001 -> PostAcc {sync dMsg_002_001!; },
Acc_002_002 -> PostAcc {sync dMsg_002_002!; },
Acc_002_003 -> PostAcc {sync dMsg_002_003!; },

PostAcc -> PostAcc_000_000 { sync nMsg_000_000?; },
PostAcc -> PostAcc_000_001 { sync nMsg_000_001?; },
PostAcc -> PostAcc_000_002 { sync nMsg_000_002?; },
PostAcc -> PostAcc_000_003 { sync nMsg_000_003?; },
```

```
  PostAcc -> PostAcc_001_000 { sync nMsg_001_000?; },
  PostAcc -> PostAcc_001_001 { sync nMsg_001_001?; },
  PostAcc -> PostAcc_001_002 { sync nMsg_001_002?; },
  PostAcc -> PostAcc_001_003 { sync nMsg_001_003?; },
  PostAcc -> PostAcc_002_000 { sync nMsg_002_000?; },
  PostAcc -> PostAcc_002_001 { sync nMsg_002_001?; },
  PostAcc -> PostAcc_002_002 { sync nMsg_002_002?; },
  PostAcc -> PostAcc_002_003 { sync nMsg_002_003?; },

  PostAcc_000_000 -> PostAcc { assign Q[0] = 0; },
  PostAcc_000_001 -> PostAcc { assign Q[0] = 1; },
  PostAcc_000_002 -> PostAcc { assign Q[0] = 2; },
  PostAcc_000_003 -> PostAcc { assign Q[0] = 3; },
  PostAcc_001_000 -> PostAcc { assign Q[1] = 0; },
  PostAcc_001_001 -> PostAcc { assign Q[1] = 1; },
  PostAcc_001_002 -> PostAcc { assign Q[1] = 2; },
  PostAcc_001_003 -> PostAcc { assign Q[1] = 3; },
  PostAcc_002_000 -> PostAcc { assign Q[2] = 0; },
  PostAcc_002_001 -> PostAcc { assign Q[2] = 1; },
  PostAcc_002_002 -> PostAcc { assign Q[2] = 2; },
  PostAcc_002_003 -> PostAcc { assign Q[2] = 3; },

  PostAcc -> PostChk_000 { guard h >= L; },
  PostChk_000 -> PostChk_001 {guard Q[0] == -1;},
  PostChk_001 -> PostChk_002 {guard Q[1] == -1;},
  PostChk_002 -> Free {guard Q[2] == -1;},
  PostChk_000 -> M2S_000_000 {guard Q[0] == 0; assign Q[0] = -1;},
  PostChk_000 -> M2S_000_001 {guard Q[0] == 1; assign Q[0] = -1;},
  PostChk_000 -> M2S_000_002 {guard Q[0] == 2; assign Q[0] = -1;},
  PostChk_000 -> M2S_000_003 {guard Q[0] == 3; assign Q[0] = -1;},
  PostChk_001 -> M2S_001_000 {guard Q[1] == 0; assign Q[1] = -1;},
  PostChk_001 -> M2S_001_001 {guard Q[1] == 1; assign Q[1] = -1;},
  PostChk_001 -> M2S_001_002 {guard Q[1] == 2; assign Q[1] = -1;},
  PostChk_001 -> M2S_001_003 {guard Q[1] == 3; assign Q[1] = -1;},
  PostChk_002 -> M2S_002_000 {guard Q[2] == 0; assign Q[2] = -1;},
  PostChk_002 -> M2S_002_001 {guard Q[2] == 1; assign Q[2] = -1;},
  PostChk_002 -> M2S_002_002 {guard Q[2] == 2; assign Q[2] = -1;},
  PostChk_002 -> M2S_002_003 {guard Q[2] == 3; assign Q[2] = -1;};
}
```

Appendix D

Using CANGen

D.1 Usage

```
    java CANGen <input file>
or  java -jar CANGen.jar <input file>
or  CANGen <input file>
```

D.2 Input Syntax

```
The first 3 lines should be
#numTypes ...    [the number of message types to be supported]
#numVals ...     [the number of message payload values to be supported]
#channel .,.,.,. [the four channel parameters in a comma-separated list]

After this, there are instances of each of the following (in no
particular order); there may be more than one instance and there
should be at least one of (b - e) -

(a) the directive
#fanoutTypes
  on a line by itself.
  This generates the process template for fanning an urgent
  synchronisation out over all types. There need only be one
  instance of this. The template signature is
```

```
process FanoutTypes(urgent chan & syn,
  urgent chan & syn000, urgent chan & syn001, urgent chan & syn002 ...)


(b) the directive
#globals
   on a line by itself, followed by lines consisting of valid Uppaal
   global variable declarations

(c) the directive
#processes
   on a line by itself, followed by lines consisting of valid Uppaal
   process (template) definitions

(d) the directive
#procInstances
   on a line by itself, followed by lines consisting of valid Uppaal
   process instantiations

(e) the directive
#sysDef
   on a line by itself, followed by a valid Uppaal system definition

There will normally be only one of (e) although there may be multiple
instances of (b-d).

Apart from these directives, CANGen input is ordinary UPPAAL .xta
sources, but there are two special syntactic construct for multiplying
elements (channel, variable declarations, locations, edges,...) over
message types and message payload values:

  FORType(t) ... token|t|... ROF
  FORVal(v)  ... token|v|... ROF

CANGen expands FORType(t) ... token|t|... ROF into a comma or semicolon
-separated list ... token000 ... , token001 ... , token002 ...
and so on up to (m-1). Every instance of |t| is replaced by a sequence
number 000, 001, ... up to (m-1); all other tokens are replicated
without substitution.
```

```
There may be several occurrences of |t| within the construct - i.e.,
between FORType(t) and ROF. A metavariable other than t may be used
but it has to match the one in FORType(.). A construct may stretch over
several lines.

UPPAAL syntax sometimes requires the multiple instances to be separated
by commas and sometimes by semicolons. CAN does whichever is suggested
by the context to be appropriate -- admittedly this is something of a
kludge.

Similarly, CANGen expands FORVal(v) ... token|v|... ROF into a comma or
semicolon-separated list ... token000 ... , token001 ... , token002 ...
and so on up to (n-1). Every instance of |v| is replaced by a sequence
number 000, 001, ... up to (n-1); all other tokens are replicated
without substitution.

The two constructs may be nested inside each other, for instance,

  FORType(t) FORVal(v) ... token|t|_|v| ... tokenB|t| ROF ROF

replicates m X n times:

... token000_000 ... tokenB000, ... token000_001 ... tokenB000,
..... ,
... token001_000 ... tokenB001, ... token001_001 ... tokenB001,
..... and so on.
```

D.3 Example

D.3.1 Input File

This is the input file for the UPPAAL system used in §6.2 to show the no-free-variables CAN channel model weakly bisimilar to the original model, in the case $m = 3$ $n = 4$.

```
#numTypes 3
#numVals 4
#channel 15, 25, 10, 15
```

```
#processes
process Sender(FORtype(t)FORval(v) urgent chan & nMsg|t|_|v|
                                                        ROFROF) {
state Init, FORtype(t)FORval(v) S|t|_|v| ROFROF;
  init Init;

trans
  FORtype(t)FORval(v)
  Init -> S|t|_|v|  { },
  S|t|_|v| -> Init  {
    sync nMsg;
  } ROFROF;
}

process Receiver(FORtype(t)FORval(v) broadcast chan & dMsg|t|_|v|
                                                          ROFROF) {
state Init, FORtype(t)FORval(v) R|t|_|v| ROFROF;
  init Init;

trans
  FORtype(t)FORval(v)
  Init -> S|t|_|v|  {
    sync dMsg;
  },
  S|t|_|v| -> Init  { } ROFROF;
}

#procInstances
sender = Sender(FORtype(t)FORval(k)newMsg_|t|_|k| ROFROF);
receiver = Receiver(FORtype(t)FORval(k)dlvrMsg_|t|_|k| ROFROF);

#sysDef
system channel, sender, receiver;
```

D.3.2 Generated UPPAAL Code

This is the code generated from the input file above for the UPPAAL system used in §6.2 to show the no-free-variables CAN channel model weakly bisimilar to the original model, in the case $m = 3$ $n = 4$.

```
const int NUMTYPES = 3;
const int NUMVALS = 4;
urgent chan
  nMsg_000_000, nMsg_000_001, nMsg_000_002, nMsg_000_003,
  nMsg_001_000, nMsg_001_001, nMsg_001_002, nMsg_001_003,
  nMsg_002_000, nMsg_002_001, nMsg_002_002, nMsg_002_003;

broadcast chan
  dMsg_000_000, dMsg_000_001, dMsg_000_002, dMsg_000_003,
  dMsg_001_000, dMsg_001_001, dMsg_001_002, dMsg_001_003,
  dMsg_002_000, dMsg_002_001, dMsg_002_002, dMsg_002_003;


process Channel(const int l, const int u,
                const int L, const int U,
```

.....This code is exactly as in appendix C.....

```
process Sender(
urgent chan & nMsg000_000 ,  urgent chan & nMsg000_001 ,
urgent chan & nMsg000_002 ,  urgent chan & nMsg000_003 ,
urgent chan & nMsg001_000 ,  urgent chan & nMsg001_001 ,
urgent chan & nMsg001_002 ,  urgent chan & nMsg001_003 ,
urgent chan & nMsg002_000 ,  urgent chan & nMsg002_001 ,
urgent chan & nMsg002_002 ,  urgent chan & nMsg002_003 ) {

state Init,  S000_000 ,  S000_001 ,  S000_002 ,  S000_003 ,
S001_000 ,   S001_001 ,  S001_002 ,  S001_003 ,  S002_000 ,
S002_001 ,  S002_002  ,  S002_003 ;
  init Init;

trans
  Init -> S000_000  { },
  S000_000 -> Init  {
    sync nMsg;
  } ,
  Init -> S000_001  { },
  S000_001 -> Init  {
    sync nMsg;
  } ,
```

```
  Init -> S000_002  { },
  S000_002 -> Init  {
    sync nMsg;
  } ,
```

.....abridged.....

```
  Init -> S002_003  { },
  S002_003 -> Init  {
    sync nMsg;
  } ;
}

process Receiver(
broadcast chan & dMsg000_000 ,  broadcast chan & dMsg000_001 ,
broadcast chan & dMsg000_002 ,  broadcast chan & dMsg000_003 ,
broadcast chan & dMsg001_000 ,  broadcast chan & dMsg001_001 ,
broadcast chan & dMsg001_002 ,  broadcast chan & dMsg001_003 ,
broadcast chan & dMsg002_000 ,  broadcast chan & dMsg002_001 ,
broadcast chan & dMsg002_002 ,  broadcast chan & dMsg002_003 ) {

state Init,  R000_000 ,  R000_001 ,  R000_002 ,  R000_003 ,
R001_000 ,   R001_001 ,  R001_002 ,  R001_003 ,  R002_000 ,
R002_001 ,  R002_002  ,  R002_003 ;
init Init;

trans
  Init -> S000_000  {
    sync dMsg;
  },
  S000_000 -> Init  { } ,
  Init -> S000_001  {
    sync dMsg;
  },
  S000_001 -> Init  { } ,
  Init -> S000_002  {
    sync dMsg;
  },
```

.....abridged.....

```
  Init -> S002_003  {
```

```
    sync dMsg;
  },
  S002_003 -> Init  { } ;
}

channel = Channel(15, 25, 10, 15,
  nMsg_000_000, nMsg_000_001, nMsg_000_002, nMsg_000_003,
  nMsg_001_000, nMsg_001_001, nMsg_001_002, nMsg_001_003,
  nMsg_002_000, nMsg_002_001, nMsg_002_002, nMsg_002_003,
  dMsg_000_000, dMsg_000_001, dMsg_000_002, dMsg_000_003,
  dMsg_001_000, dMsg_001_001, dMsg_001_002, dMsg_001_003,
  dMsg_002_000, dMsg_002_001, dMsg_002_002, dMsg_002_003);


sender = Sender(newMsg_000_000 , newMsg_000_001 ,
newMsg_000_002 , newMsg_000_003 , newMsg_001_000,
newMsg_001_001 , newMsg_001_002 , newMsg_001_003,
newMsg_002_000 , newMsg_002_001 , newMsg_002_002,
newMsg_002_003 );
receiver = Receiver(dlvrMsg_000_000 , dlvrMsg_000_001,
dlvrMsg_000_002 , dlvrMsg_000_003 , dlvrMsg_001_000 ,
dlvrMsg_001_001 , dlvrMsg_001_002 , dlvrMsg_001_003 ,
dlvrMsg_002_000 , dlvrMsg_002_001 , dlvrMsg_002_002 ,
dlvrMsg_002_003 );

system channel, sender, receiver;
```

Appendix E

A First Example – source

E.1 System with a single sensor

```
#numTypes 3
#numVals 40
#channel 15, 25, 10, 15

#processes
process PlantSensor(int p,
  broadcast chan & swOff, broadcast chan & swOn,  broadcast chan & rq,
  FORval(k) urgent chan &rdg|k|ROF) {

clock g, h;
bool on;
int[0,50] v;

state ini, p1{h <= p}, rqstd, FORval(k)val|k|ROF;
  urgent ini, rqstd, FORval(k)val|k|ROF;
  init ini;

trans
  ini -> p1  {
    assign g = 0, h = 0, v = NUMVALS/2, on = false;
  },
  p1 -> p1   {
    sync swOn?;
```

```
  assign on = true;
},
p1 -> p1   {
  sync swOff?;
  assign on = false;
},
p1 -> p1   {
  guard h >= p/2 && on;
  assign v ++, h = 0;
},
p1 -> p1   {
  guard h >= p/2 && !on;
  assign v --, h = 0;
},
p1 -> rqstd {
  sync rq?;
},
rqstd -> rqstd {
  sync swOn?;
  assign on = true;
},
rqstd -> rqstd {
  sync swOff?;
  assign on = false;
},
FORval(k)
rqstd -> val|k| {
  guard v == |k|;
},
val|k| -> val|k| {
  sync swOn?;
  assign on = true;
},
val|k| -> val|k| {
  sync swOff?;
  assign on = false;
},
val|k| -> p1 {
  sync rdg|k|!;
}ROF;
```

```
}

process Controller(int hi, int lo, int dly, int dly2,
  urgent chan & swOff, urgent chan & swOn, urgent chan & rq,
  FORval(k) broadcast chan &rdg|k|ROF) {

clock g, h;
int[0,50] n, vAv;

state
  ini, rqstVal{h<=dly}, rqstd, getVals{h<=dly2},
    sndOn{h<=dly2}, sndOff{h<=dly2},
    FORval(k)got|k|ROF;
  urgent ini, rqstd,
    FORval(k)got|k|ROF;
  init ini;

trans
  ini -> rqstVal {
    assign vAv = 0, g = 0, h=0;
  },
  rqstVal -> rqstd {
    guard h >= dly;
    assign h = 0;
  },
  rqstd -> getVals {
    sync rq!;
    assign h=0;
  },
  FORval(k)
  getVals -> got|k| {
    sync rdg|k|?;
    assign vAv = |k|;
  },
  got|k| -> rqstVal {
    guard vAv <= hi && vAv >= lo;
    assign h = 0;
  },
  got|k| -> sndOff {
    guard vAv > hi;
    assign h = 0;
```

```
  },
  got|k| -> sndOn {
    guard vAv < lo;
    assign h = 0;
  }ROF,
  sndOn -> rqstVal {
    sync swOn!;
    assign h = 0;
  },
  sndOff -> rqstVal {
    sync swOff!;
    assign h = 0;
  },
  getVals -> getVals {
    guard h >= dly2;
    assign h=0;
  };
}

#procInstances
plantSensor = PlantSensor(2000, dMsg_001_000, dMsg_001_001,
   dMsg_002_000, FORval(k)nMsg_000_|k| ROF);

controller = Controller(21, 19, 5, 5, nMsg_001_000, nMsg_001_001,
  nMsg_002_000, FORval(k)dMsg_000_|k| ROF);

#sysDef
system channel, plantSensor, controller;
```

E.2 System which averages two sensors

This is identical to the 1-sensor version (globals, network specification, controller process, sensor process signature, process instantiation, system specification) *except* for the definition of the sensor process, which now carries two sensed variables and averages them:

```
process PlantSensor(int p,
  urgent chan & swOff, urgent chan & swOn,  urgent chan & rq,
```

```
  FORval(k) urgent chan &rdg|k|ROF) {

clock g, h;
bool on;
int[0,50] v1, v2;

state ini, p1{h <= p}, rqstd, FORval(k)val|k|ROF;
  urgent ini, rqstd, FORval(k)val|k|ROF;
  init ini;

trans
  ini -> p1  {
    assign g = 0, h = 0, v1 = NUMVALS/2, v2 = NUMVALS/2,
    on = false;
  },
  p1 -> p1   {
    sync swOn?;
    assign on = true;
  },
  p1 -> p1   {
    sync swOff?;
    assign on = false;
  },
  p1 -> p1   {
    guard h >= p/2 && on;
    assign v1 ++, h = 0;
  },
  p1 -> p1   {
    guard h >= p/2 && on;
    assign v2 ++, h = 0;
  },
  p1 -> p1   {
    guard h >= p/2 && !on;
    assign v1 --, h = 0;
  },
  p1 -> p1   {
    guard h >= p/2 && !on;
    assign v2 --, h = 0;
  },
  p1 -> rqstd {
    sync rq?;
```

```
  },
  rqstd -> rqstd {
    sync swOn?;
    assign on = true;
  },
  rqstd -> rqstd {
    sync swOff?;
    assign on = false;
  },
  FORval(k)
  rqstd -> val|k| {
    guard (v1+v2)/2 == |k|;
  },
  val|k| -> val|k| {
    sync swOn?;
    assign on = true;
  },
  val|k| -> val|k| {
    sync swOff?;
    assign on = false;
  },
  val|k| -> p1 {
    sync rdg|k|!;
  }ROF;
}
```

Appendix F

Using Assume-Guarantee – sources

F.1 The concrete system model

```
#numTypes 3
#numVals 40
#channel 15, 25, 10, 15

#processes
process PlantSensor(int p, int in,
  broadcast chan & swOff, broadcast chan & swOn,
  broadcast chan & rq,
  FORval(k) urgent chan &rdg|k|ROF) {

clock g, h;
bool on;
int[0,50] v;

state ini, p1{h <= p}, rqstd, FORval(k)val|k|ROF;
  urgent ini, rqstd, FORval(k)val|k|ROF;
  init ini;

trans
  ini -> p1  {
```

```
  assign g = 0, h = 0, v = NUMVALS/2, on = false;
},
p1 -> p1   {
  sync swOn?;
  assign on = true, g = 0;
},
p1 -> p1   {
  sync swOff?;
  assign on = false, g = 0;
},
p1 -> p1   {
  guard h >= p/2 && on && g > in;
  assign v ++, h = 0;
},
p1 -> p1   {
  guard h >= p/2 && !on && g > in;
  assign v --, h = 0;
},
p1 -> p1   {
  guard h >= p/2 && g <= in;
  assign h = 0;
},
p1 -> rqstd {
  sync rq?;
},
rqstd -> rqstd {
  sync swOn?;
  assign on = true, g = 0;
},
rqstd -> rqstd {
  sync swOff?;
  assign on = false, g = 0;
},
FORval(k)
rqstd -> val|k| {
  guard v == |k|;
},
val|k| -> val|k| {
  sync swOn?;
  assign on = true, g = 0;
},
```

```
  val|k| -> val|k| {
    sync swOff?;
    assign on = false, g = 0;
  },
  val|k| -> p1 {
    sync rdg|k|!;
  }ROF;

}

process Controller(int hi, int lo, int e, int dly, int dly2,
  int inert, urgent chan & swOff, urgent chan & swOn,
  urgent chan & rq, FORval(k) broadcast chan &rdg|k|ROF) {

clock g, h;
int[0,50] vAv;

state
  ini, rqstVal{h<=dly}, rqstd, getVals,
    sndOn, sndOff,
    FORval(k)got|k|{h<=dly2}ROF;
  urgent ini;
  init ini;

trans
  ini -> rqstVal {
    assign vAv = NUMVALS/2, h=0, g=0;
  },
  rqstVal -> rqstd {
    guard h >= dly;
    assign h = 0;
  },
  rqstd -> getVals {
    sync rq!;
    assign h=0;
  },
  FORval(k)
  getVals -> got|k| {
    sync rdg|k|?;
    assign vAv = |k|, h = 0;
  },
```

```
  got|k| -> rqstVal {
    guard (vAv <= hi && vAv >= lo) || (g <= inert
                && (vAv <= hi+e && vAv >= lo-e));
    assign h = 0;
  },
  got|k| -> sndOff {
    guard (vAv > hi && g > inert) || vAv > hi+e;
    assign h = 0;
  },
  got|k| -> sndOn {
    guard (vAv < lo && g > inert) || vAv < lo-e;
    assign h = 0;
  }ROF,
  sndOn -> rqstVal {
    sync swOn!;
    assign h = 0, g = 0;
  },
  sndOff -> rqstVal {
    sync swOff!;
    assign h = 0, g = 0;
  };
}

#procInstances
plantSensor = PlantSensor(10000, 50, dMsg_001_000, dMsg_001_001,
  dMsg_002_000, FORval(k)nMsg_000_|k| ROF);

controller = Controller(21, 19, 2, 5, 5, 100, nMsg_001_000,
  nMsg_001_001, nMsg_002_000, FORval(k)dMsg_000_|k| ROF);

#sysDef
system channel, plantSensor, controller;
```

F.2 The abstraction

```
.................
process PlantSensor(int p,
  broadcast chan & swOff, broadcast chan & swOn,
```

```
  broadcast chan & rq,
  FORval(k) urgent chan &rdg|k|ROF) {

clock h;
bool on;
int[0,50] v;

state ini, p1{h <= p}, rqstd, FORval(k)val|k|ROF;
  urgent ini, rqstd, FORval(k)val|k|ROF;
  init ini;

trans
  ini -> p1  {
    assign h = 0, v = NUMVALS/2, on = false;
  },
  p1 -> p1   {
    sync swOn?;
    assign on = true;
  },
  p1 -> p1   {
    sync swOff?;
    assign on = false;
  },
  p1 -> p1   {
    guard h >= p/2 && on;
    assign v ++, h = 0;
  },
  p1 -> p1   {
    guard h >= p/2 && !on;
    assign v --, h = 0;
  },
  p1 -> p1   {
    guard h >= p/2;
    assign h = 0;
  },
  p1 -> rqstd {
    sync rq?;
  },
  rqstd -> rqstd {
    sync swOn?;
    assign on = true;
```

```
  },
  rqstd -> rqstd {
    sync swOff?;
    assign on = false;
  },
  FORval(k)
  rqstd -> val|k| {
    guard v == |k|;
  },
  val|k| -> val|k| {
    sync swOn?;
    assign on = true;
  },
  val|k| -> val|k| {
    sync swOff?;
    assign on = false;
  },
  val|k| -> p1 {
    sync rdg|k|!;
  }ROF;

}

process Controller(int hi, int lo, int e, int dly, int dly2,
  urgent chan & swOff, urgent chan & swOn, urgent chan & rq,
  FORval(k) broadcast chan &rdg|k|ROF) {

clock h;
int[0,50] vAv;

state
  ini, rqstVal{h<=dly}, rqstd, getVals,
    sndOn, sndOff,
    FORval(k)got|k|{h<=dly2}ROF;
  urgent ini;
  init ini;

trans
  ini -> rqstVal {
    assign vAv = NUMVALS/2, h=0;
  },
```

```
  rqstVal -> rqstd {
    guard h >= dly;
    assign h = 0;
  },
  rqstd -> getVals {
    sync rq!;
    assign h=0;
  },
  FORval(k)
  getVals -> got|k| {
    sync rdg|k|?;
    assign vAv = |k|, h = 0;
  },
  got|k| -> rqstVal {
    guard vAv <= hi+e && vAv >= lo-e;
    assign h = 0;
  },
  got|k| -> sndOff {
    guard vAv > hi;
    assign h = 0;
  },
  got|k| -> sndOn {
    guard vAv < lo;
    assign h = 0;
  }ROF,
  sndOn -> rqstVal {
    sync swOn!;
    assign h = 0;
  },
  sndOff -> rqstVal {
    sync swOff!;
    assign h = 0;
  };
}

#procInstances
plantSensor = PlantSensor(10000, dMsg_001_000, dMsg_001_001,
  dMsg_002_000, FORval(k)nMsg_000_|k| ROF);

controller = Controller(21, 19, 2, 5, 5, nMsg_001_000,
  nMsg_001_001, nMsg_002_000, FORval(k)dMsg_000_|k| ROF);
```

```
#sysDef
system channel, plantSensor, controller;
```