An Integration of Real-Time Object-Z and CSP for Specifying Concurrent Real-Time Systems

Graeme Smith

Software Verification Research Centre, University of Queensland, Australia
smith@svrc.uq.edu.au

Abstract. Real-Time Object-Z is an integration of the object-oriented formal specification language Object-Z with a timed trace notation suitable for modelling timing constraints and continuous variables. This extends the applicability of Object-Z to real-time and embedded systems. In this paper, we enhance the ability of Real-Time Object-Z to specify concurrent real-time and embedded systems by semantically integrating it with the process algebra CSP. The approach builds on the existing work on the integration of (standard) Object-Z and CSP.

1 Introduction

Object-Z [16,3] is an object-oriented specification language based on Z [21]. It extends Z with a notion of *classes*, used to encapsulate a state schema with its initial state schema and associated set of operations, and *objects*, instances of classes used to specifying systems. The enhanced structuring provided by classes and associated techniques such as *inheritance*, which enables definitions of one class to include those of another, and *polymorphism*, which enables the construction of a type corresponding to a collection of classes, makes Object-Z well-suited to modelling large-scale systems with complex data structures.

When modelling interaction in concurrent systems, however, Object-Z specification can become unwieldy. This is due to the necessity to explicitly specify all concurrent occurrences of operations [16, Chapter 5]. This shortcoming led to the development of Object-Z/CSP [15,17,2,18], a semantic integration[1] of Object-Z and CSP [9,12] in which Object-Z classes are identified with CSP processes so that interaction between instances of them can be specified using CSP operators.

In order to model real-time and embedded systems, Object-Z has also been semantically integrated with the timed trace notation of Fidge et al. [5] in which variables are modelled as (possibly continuous) functions defining their values over all time. The integrated notation, referred to as Real-Time Object-Z [19,20], suffers from the same shortcomings with respect to modelling concurrent systems as standard Object-Z.

[1] A semantic integration is one in which language constructs of the constituent languages are semantically identified and usually involves no change to the syntax of either language [6].

M. Butler, L. Petre, and K. Sere (Eds.): IFM 2002, LNCS 2335, pp. 267–285, 2002.

To overcome this problem with Real-Time Object-Z, Smith and Hayes [20], developed a simple parallel composition operator. This operator, which identifies common-named inputs, outputs and operations of instances of combined classes, provides a concise means of modelling concurrency. However, it is overly restrictive since it requires synchronising operations to have the same start and end times. Furthermore, it provides only a limited means for constructing concurrent systems. For example, there is no support for combining instances of classes which do not synchronise on common-named operations nor any means of renaming or hiding operations.

In this paper, we provide an alternative approach to concurrency in Real-Time Object-Z. Building on the work on Object-Z/CSP, we semantically identify Real-Time Object-Z classes with CSP processes allowing instances of them to be combined with the full range of CSP operators. We begin by providing an overview of Object-Z/CSP (in Section 2) and Real-Time Object-Z (in Section 3). We then show how a failures/divergences semantics (the semantics of CSP processes) can be given to Real-Time Object-Z classes by associating CSP events with (instantaneous) observations of operations and process parameters with continuous variables (in Section 4). We discuss alternative approaches for specifying real-time concurrent systems based on Object-Z (in Section 5) before concluding (in Section 6).

2 Overview of Object-Z/CSP

Object-Z/CSP [15,17,2,18] is an integration of Object-Z [16,3] and CSP [9,12] motivated by the need to model both complex data structures and process interaction in the specification of concurrent and distributed systems. Object-Z classes are used to model data structures, comprising a state and collection of operations, and CSP operators are used to model the interactions between instances of these classes.

The integration is *semantic* in the sense that a common semantics is given to the two languages and hence to the overall specifications. This approach has two main advantages. Firstly, the individual notations are syntactically unchanged and clearly separated in specifications. This makes the specifications more accessible to users already familiar with the languages, and the use of existing tool support and verification and refinement methods possible [8,17,18]. Secondly, there is no need to define a new semantics for the integration. Adopting the existing semantics of one of the languages is possible. The semantics adopted for Object-Z/CSP is the existing failures/divergences semantics of CSP.

To illustrate Object-Z/CSP, we specify a simple case study. The case study is based on a proposed system to help farmers in outback Australia keep track of the condition of their cattle[2]. Due to the large size of cattle properties, it is not always possible for farmers to be aware of the condition of their cattle, and hence to know whether additional food needs to be brought to them in

[2] The author worked on the implementation of this system while an undergraduate student. He is not aware whether it is still in use.

times of drought. To overcome this, weighbridges can be installed around fenced waterholes. As the cattle enter and leave the area around the waterhole, their weight is recorded and trends in weight loss or gain can be noticed by the farmer.

Typically, the system would comprise several weighbridges, each with its own *weigh unit*, and a central *store unit* (see Fig. 1).

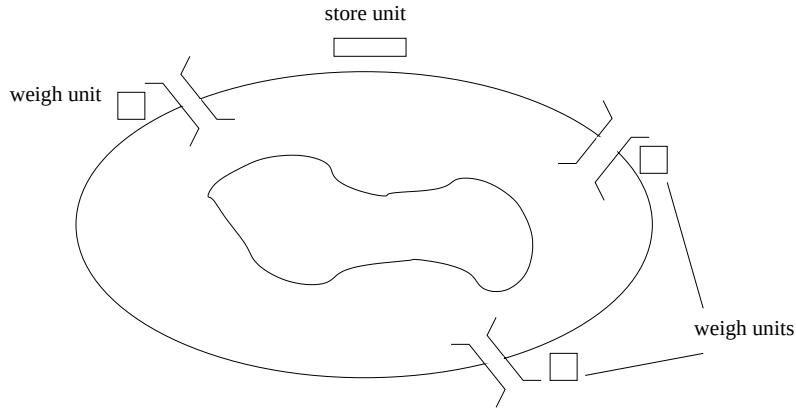

Fig. 1. Aerial view of cattle weighing system

The weigh units calculate the weight of individual cattle from a continuous signal input from the weighbridge (see Fig. 2).

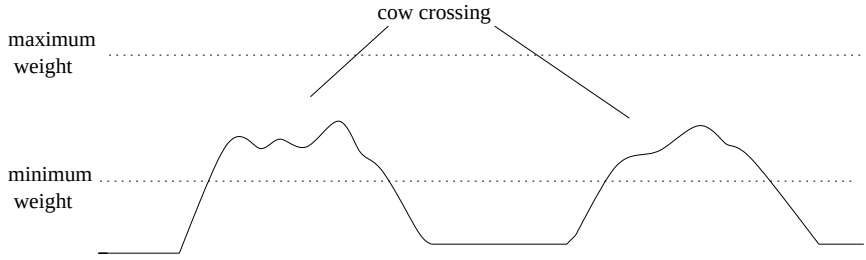

Fig. 2. Typical weighbridge signal

These weights are then transmitted to the store unit which stores them along with the day they were recorded.

To specify such a system in Object-Z/CSP, we would begin by specifying its components, weigh units and store units, as Object-Z classes. As a preliminary, we specify the set of non-negative real numbers $\mathbb{R}_+$ for representing weights and the constants *MinWeight* and *MaxWeight* denoting the minimum weight that

should be detected as a cattle crossing and the maximum weight the system needs to be able to deal with respectively.

$$\mathbb{R}_+ == \{r : \mathbb{R} \mid r \geqslant 0\}$$

$$MinWeight, MaxWeight : \mathbb{R}_+$$
$$MinWeight < MaxWeight$$

A weigh unit is specified as having a single state variable: *data* comprising recently recorded data not yet transmitted to the store unit. Initially, there is no recorded data. An operation *Weigh* allows a new weight, input as *weight*?, to be added to the data, and an operation *SendData* allows the recorded data to be output, as *data*!, and cleared. Note that the input *weight*? of the former operation is an abstraction of the actual continuous signal from the weighbridge. It (informally) corresponds to the average value of the weighbridge signal over an interval of time where the signal is greater than the level "minimum weight" (see Fig. 2).

WeighUnit

$$data : \mathrm{bag}\, \mathbb{R}_+$$

INIT

$$data = [\![\,]\!]$$

Weigh

$$\Delta(data)$$
$$weight? : \mathbb{R}_+$$
$$MinWeight \leqslant weight? \leqslant MaxWeight$$
$$data' = data \uplus [\![weight?]\!]$$

SendData

$$\Delta(data)$$
$$data! : \mathrm{bag}\, \mathbb{R}_+$$
$$data! = data$$
$$data' = [\![\,]\!]$$

A store unit is specified as having two variables: *day* denoting the day of operation of the system (incremented every 24 hours), and *record* denoting the data received from the weigh units each day. Initially, *day* is set to 1 and there is no received data. An operation *RecData* allows data to be received and added to any other data received for that day. An operation *UpdateDay* allows the day to be incremented.

StoreUnit

$$day : \mathbb{N}$$
$$record : \mathbb{N} \rightarrow \text{bag}\, \mathbb{R}_+$$

INIT

$$day = 1$$
$$\forall\, n : \mathbb{N} \bullet record(n) = [\![\,]\!]$$

RecData

$$\Delta(data)$$
$$data? : \text{bag}\, \mathbb{R}_+$$
$$record' = record \oplus \{day \mapsto record(day) \uplus data?\}$$

UpdateDay

$$\Delta(day)$$
$$day' = day + 1$$

To specify systems of components, Object-Z/CSP views such classes as processes which can be combined using the operators of CSP. In particular, operations are identified with events and, to synchronise, must have the same name and parameters with the same basenames (i.e., apart from the ? or !) and values. In the case study, we want *RecData* of *StoreUnit* to synchronise with *SendData* of *WeighUnit*. Hence, we specify new classes *SU* and *WU* which inherit *StoreUnit* and *WeighUnit*, respectively, applying appropriate renaming.

SU

$$StoreUnit[ComData/RecData]$$

We also need to refer to multiple instances of *WeighUnit*. This is facilitated by adding a constant to the class so that distinct weigh units can be assigned unique numbers. Given the specification below, $WU_{\{number \mapsto i\}}$ denotes an instance of *WU* with $number = i$ [15].

WU

$$WeighUnit[ComData/SendData]$$
$$number : \mathbb{N}$$

The weighing system is specified as a parameterised system. The parameter denotes how many weigh units are present. The weigh units are combined with each other using the CSP interleaving operator ||| (i.e., they do not synchronise with each other on any events) and with the store unit using the CSP parallel operator ‖ (i.e., the store unit performs the (renamed) operation *SendData* only when one of the weigh units can synchronise with it).

$$WeighingSystem(n) = (|||_{i=1}^{n}\ WU_{\{number \mapsto i\}})\ {}_X\|_Y\ SU$$

where $X = \{|\ Weigh, SendData\ |\}$ and $Y = \{|\ SendData, UpdateDay\ |\}$.

The preceding specification of the cattle weighing system has several shortcomings. Firstly, the interpretation of the *weight*? inputs is informal. Additionally, the fact that the *Weigh* operations occur every time their associated weighbridge signals are greater than the level "minimum weight", is not specified.

There are also some timing constraints which are not formalised. Firstly, the fact the *UpdateDay* operation occurs regularly at an appropriate time is not formalised. Secondly, the fact that the information recorded by the weigh units is transmitted to the store unit on the same day they are recorded is similarly omitted. Both of these timing constraints are crucial to the information in the store unit being of use to the farmer.

3 Overview of Real-Time Object-Z

Real-Time Object-Z [19,20] is an integration of Object-Z with the timed trace notation of Fidge et al. [5]. It allows the specification of complex data structures which behave according to real-time constraints and interact with a continuously changing environment. A type $\mathbb{T} == \mathbb{R}$ is introduced to model absolute time. In this paper, we assume its units are seconds and use constants min=60 and hour=60*60 to allow us to write times in minutes or hours respectively.

Classes are divided into two parts by a horizontal line. The part above the line is essentially standard Object-Z with the addition of an implicit variable τ of type $\mathbb{T}$ denoting the current time. The part below the line contains two timed trace predicates denoting an assumption on the class's environment and an effect the class achieves when that assumption is met.

Using Real-Time Object-Z, the class *WeighUnit* can be modified as follows. A timed trace constant *weight*?, denoting the continuous signal from the weighbridge over all time, replaces the need for the input *weight*? of the operation *Weigh*. The ? decoration on this constant denotes that it is an input from the environment (a ! decoration would similarly denote an output). The fact that such an input is continuous (and smooth) is specified using the function symbol $\rightsquigarrow$ [4]. A timed trace assumption predicate is added to the class restricting the value of *weight*? to always be less than or equal to *MaxWeight*.

The operation *Weigh* uses this environmental input to calculate the weight of a cow. It adds to *data* the average value of the signal between its start time τ and end time τ' [3]. To ensure this operation occurs when necessary (and not otherwise), we add an effect predicate which states that the operation occurs precisely when *weight*? is at least *MinWeight*.

The predicate uses the notation $\langle P \rangle$ to denote the set of intervals of time in which a predicate P holds. In general, any timed trace constants and variables

[3] In practice, some error, due to sampling and time delays, would be introduced in the calculation of the average weight. For simplifying the presentation, however, we ignore this and other such errors in this paper.

in such a predicate P may be *lifted* to their range types [5], i.e., a constant or variable of type $\mathbb{T} \to T$ is treated as if it were a variable of type T (e.g., *weight*? in the predicate below). Such a predicate P may also include the names of operations denoting Boolean variables which are true precisely when the operation is occurring (e.g., *Weigh* in the predicate below).

The frequency of occurrence of the operation *SendData* is also constrained by a timed trace effect predicate. This predicate use the operator ';' for concatenating sets of time intervals [5]. It specifies that in any time interval of at least 10 minutes duration, the operation *SendData* must occur. This is done using the reserved symbol δ which denotes the duration of an interval. Similarly, α and ω are reserved symbols denoting the start and end times of an interval, and ϕ is a reserved symbol denoting the interval itself.

WeighUnit

$weight? : \mathbb{T} \rightsquigarrow \mathbb{R}_+$

$data : \operatorname{bag} \mathbb{R}_+$

INIT

$data = \langle\,\rangle$

Weigh

$\Delta(data)$

$data' = data \uplus [\![(\int_{\tau}^{\tau'} weight?)/(\tau' - \tau)]\!]$

SendData

$\Delta(data)$
$data! : \operatorname{bag} \mathbb{R}_+$

$data! = data$
$data' = [\![\,]\!]$

---

assumption $\forall\, t : \mathbb{T} \bullet weight?(t) \leqslant MaxWeight$

effect $\langle Weigh \rangle = \langle weight? \geqslant MinWeight \rangle$
$\langle \delta \geqslant 10 * \min \rangle \subseteq \langle true \rangle \,;\, \langle SendData \rangle \,;\, \langle true \rangle$

The class *StoreUnit* is extended with a variable *last_update* denoting the last time the day was updated. Initially, this value is equal to the current time. The operation *UpdateDay* is extended to set this variable to its start time and to only occur when this time is 24 hours since the time held by the variable.

By itself, the precondition of *UpdateDay* only prevents the operation from happening at times other than 24 hours after *last_update*. It does not ensure that the operation occurs when its precondition is satisfied. To specify this, we

add an effect predicate which states that the operation occurs in every 24 hour interval of time.

StoreUnit

$$day : \mathbb{N}$$
$$record : \mathbb{N} \rightarrow \text{bag}\,\mathbb{R}_+$$
$$last\_update : \mathbb{T}$$

INIT

$$day = 1$$
$$\forall\, n : \mathbb{N} \bullet record(n) = [\![\,]\!]$$
$$last\_update = \tau$$

RecData

$$\Delta(data)$$
$$data? : \text{bag}\,\mathbb{R}_+$$
$$record' = record \oplus \{day \mapsto record(day) \uplus data?\}$$

UpdateDay

$$\Delta(day, last\_update)$$
$$\tau - last\_update = 24 * \text{hour}$$
$$day' = day + 1$$
$$last\_update' = \tau$$

assumption *true*

effect $\langle \delta = 24 * \text{hour} \rangle \subseteq \langle true \rangle \,;\, \langle UpdateDay \rangle \,;\, \langle true \rangle$

The above classes overcome the shortcomings identified at the end of Section 2. The interpretation of the input to the *Weigh* operation is formalised as are the constraints on the occurrence of *Weigh*, *UpdateDay* and *SendData*. To specify the weighing system, however, we would like to be able to combine these classes in a manner similar to that in Section 2.

4 Semantic Integration of Real-Time Object-Z and CSP

To enable instances of Object-Z classes to be combined with CSP operators in Object-Z/CSP, classes are given a failures/divergences semantics, i.e., the semantics of processes in CSP. This semantics is derived from the existing history semantics of Object-Z [14].

A history of a class is a possible sequence of states an instance of the class can pass through, together with the associated sequence of operations that cause the state changes. A state is an assignment of values to a set of identifiers rep-

resenting its variables and the constants it can refer to. The states S of a class are hence defined as

$$S \subseteq Id \twoheadrightarrow Value$$

An operation comprises the operation's name and an assignment of values to the operations parameters. The operations O of a class are defined as

$$O \subseteq Id \times (Id \twoheadrightarrow Value)$$

Therefore, the set of histories of a class is represented by a set[4]

$$H \subseteq S^\omega \times O^\omega$$

such that

$$(s, o) \in H \Rightarrow s \neq \langle\,\rangle \tag{H1}$$
$$(s, o) \in H \wedge s \in S^* \Rightarrow \#s = \#o + 1 \tag{H2}$$
$$(s, o) \in H \wedge s \notin S^* \Rightarrow o \notin O^* \tag{H3}$$
$$(s_1 \frown s_2, o_1 \frown o_2) \in H \wedge \#s_1 = \#o_1 + 1 \Rightarrow (s_1, o_1) \in H \tag{H4}$$

These properties capture the fact that the sequence of states is non-empty (H1) and is one longer than the sequence of operations (H2) (except when both are infinite (H3)), and that the set of histories is prefix-closed (H4).

To relate Object-Z classes and CSP processes, we identify Object-Z operations with CSP events. In order that common-named operations synchronise and communicate via parameters with common basenames, we define a function *event* which, given an operation (n, p), where n is the operation's name and p is an assignment of values to its parameters, returns the event $n.p'$ where p' is p with all parameters replaced by their basenames (i.e., with the ? and ! decorations removed).

The failures of an Object-Z class C with constants $\vec{c}$ assigned values $\vec{v}$ are then derived from its histories as follows: (tr, X) is a failure of $C_{\{\vec{c} \mapsto \vec{v}\}}$ if

- there exists a finite history of C whose initial state is satisfied by the assignment of $\vec{v}$ to $\vec{c}$,
- the sequence of operations of the history correspond to the sequence of events in tr, and
- for each event in X, there does not exist a history which extends the original history by an operation corresponding to the event.

$$\begin{aligned} failures(C_{\{\vec{c} \mapsto \vec{v}\}}) = \{(tr, X) \mid \exists (s, o) \in H \bullet \\ s \in S^* \wedge \\ \{\vec{c} \mapsto \vec{v}\} \subseteq s(1) \wedge \\ \#tr = \#o \wedge \\ (\forall\, i \in 1 \,..\, \#tr \bullet tr(i) = event(o(i))) \wedge \\ (\forall\, e \in X \bullet (\nexists st \in S, op \in O \bullet \\ \qquad e = event(op) \wedge (s \frown \langle st \rangle, o \frown \langle op \rangle) \in H))\} \end{aligned}$$

[4] S^ω and S^* denote the set of (possibly infinite) sequences and set of finite sequences, respectively, of elements from the set S.

This definition assumes classes do not have liveness constraints (since it only refers to finite histories) and that outputs of class instances are angelic, i.e., for the purpose of specification they can be influenced by the environment [15]. Alternative semantics also exist which support liveness constraints [7] and demonic outputs [18]. Operations are also assumed to be *blocked* outside their preconditions, i.e., they do not behave chaotically. Hence, the set of divergences of a class instance $C_{\{\vec{c}\mapsto\vec{v}\}}$ is empty.

$$divergences(C_{\{\vec{c}\mapsto\vec{v}\}}) = \varnothing$$

Problems due to unbounded nondeterminism are avoided by restricting hiding of unbounded sequences of events [15].

To similarly integrate Real-Time Object-Z with CSP, we need means of encoding both timing constraints and continuous variables in CSP process definitions. There are a number of approaches to the former. For example, start and end times could be associated with events. That is, an operation Op starting at time t_1 and ending at time t_2 could be represented by an event $Op.t_1.t_2$. Alternatively, the operation could be associated with two events, $Op_S.t_1$ and $Op_E.t_2$, denoting the start and end of the operation respectively.

The problem with these approaches is that they restrict the way in which synchronisations can occur between processes. The first approach forces two synchronising events to have the same start and end times (since they have to agree on the values t_1 and t_2). The second approach forces synchronisation to occur at the start or end times of the operations, or at the start time of one and the end time of the other. In general, synchronising operations may simply overlap and not have any common start or end times (see Fig. 3).

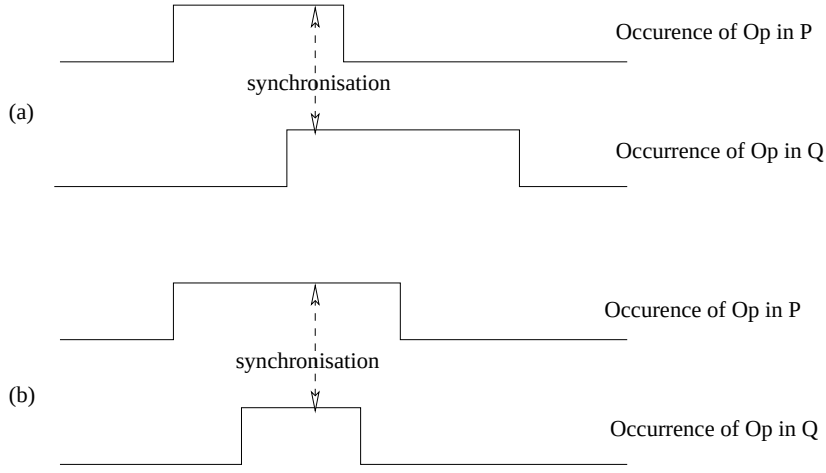

Fig. 3. Possible operation synchronisations

A more general approach, which we adopt in this paper, is to represent an operation occurrence by a single event and a single time during the occurrence.

In this case, the event corresponds to an instantaneous observation of the time-consuming operation. For example, if an operation *Op* starts at time 10 and ends at time 15, it is represented by a single event $Op.t$ where $10 \leqslant t \leqslant 15$. This enables operations to synchronise whenever they overlap, i.e., have at least one point in time in common.

Since continuous variables in Real-Time Object-Z are class constants, they can be assigned values when an instance of the class is used in a specification (cf., the constant *number* of class *WU* in Section 2). The resulting failures of the class instance depend on the value assigned. For example, the times of events related to the operation *Weigh* of *WeighUnit* depend directly on the value assigned to the continuous variable *weight*? (see Fig. 4).

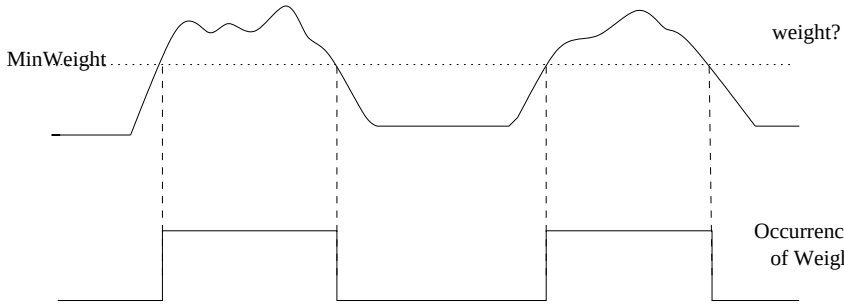

Fig. 4. Dependence of *Weigh* operations on *weight*?

When a class instance is used in a specification, the values of any continuous variables need to be supplied as additional parameters. For example, given the definitions of *WeighUnit* and *StoreUnit* of Section 3 together with the definitions of *WU* and *SU* of Section 2, the weighing system can be specified as

$$WeighingSystem(n, w_1, \ldots, w_n) = (|||_{i=1}^{n}\ WU_{\{weight? \mapsto w_i, number \mapsto i\}})\ {}_X\|_Y\ SU$$

where $X = \{|\ Weigh, SendData\ |\}$ and $Y = \{|\ SendData, UpdateDay\ |\}$.

To formalise this approach, we need to translate the semantics of Real-Time Object-Z to appropriate failures and divergences. The semantics as given by Smith and Hayes [19,20] models a class as a set of *real-time histories*. A real-time history extends a standard Object-Z history with

- start and end times of each operation,
- timed trace representations of all constants and variables, and
- a set of time intervals for each operation denoting the operation occurrences.

Since the latter can be derived from the start and end times of operations [19,20], we do not need to include them explicitly as part of the semantics. Similarly, since the timed trace representation of constants and variables can be derived from the sequence of states [19,20][5], we do not need to explicitly include them either.

[5] Note that since continuous variables are modelled as constants, their value (over all time) is available in any state.

The start times are represented by a sequence of times equal in length to the number of operations (or infinite when the number of operations are infinite). Similarly, the end times are represented by a sequence of times. The first end time denotes the time at which initialisation occurred. Hence, the length of the sequence is one greater than the number of start times (or infinite when the number of start times is infinite).

Therefore, the real-time histories of a class are represented by a set

$$R \subseteq S^\omega \times O^\omega \times \mathbb{T}^\omega \times \mathbb{T}^\omega$$

such that

$$(s, o, t_s, t_e) \in R \Rightarrow s \neq \langle\,\rangle \wedge (\forall\, i \in 1\,..\,\#t_s \bullet t_e(i) \leqslant t_s(i) \leqslant t_e(i+1)) \quad \text{(R1)}$$
$$(s, o, t_s, t_e) \in R \wedge s \in S^* \Rightarrow \#s = \#o + 1 = \#t_s + 1 = \#t_e \quad \text{(R2)}$$
$$(s, o, t_s, t_e) \in R \wedge s \notin S^* \Rightarrow o \notin O^* \wedge t_s \notin \mathbb{T}^* \wedge t_e \notin \mathbb{T}^* \quad \text{(R3)}$$
$$(s_1 \frown s_2, o_1 \frown o_2, t_{s1} \frown t_{s2}, t_{e1} \frown t_{e2}) \in R$$
$$\wedge\, \#s_1 = \#o_1 + 1 = \#t_{s1} + 1 = \#t_{e1} \Rightarrow (s_1, o_1, t_{s1}, t_{e1}) \in R \quad \text{(R4)}$$

These properties extend those for standard Object-Z histories so that there is an appropriate ordering on start and end times of operations (R1), and the sequences of start and end times are of the same length as the sequence of operations, and one more than the sequence of operations, respectively (R2) (except when each of the sequences are infinite (R3)).

The integration of Real-Time Object-Z and CSP is formalised via the following derivations of failures. (The set of divergences of a class is empty as for Object-Z/CSP. Problems with unbounded nondeterminism are avoided by an identical restriction on hiding.)

(tr, X) is a failure of $C_{\{\vec{c} \mapsto \vec{v}\}}$ if

- there exists a finite real-time history of C whose initial state is satisfied by the assignment of $\vec{v}$ to $\vec{c}$,
- each event in tr represents the corresponding operation in the history together with a time of occurrence between the corresponding start and end times, and
- for each event in X, there does not exist a real-time history which extends the original real-time history by an operation corresponding to the event.

$$\begin{aligned}
\mathit{failures}_{rt}(C_{\{\vec{c} \mapsto \vec{v}\}}) = \{(tr, X) \mid \exists (s, o, t_s, t_e) \in R \bullet \\
s \in S^* \wedge \\
\{\vec{c} \mapsto \vec{v}\} \subseteq s(1) \wedge \\
\#tr = \#o \wedge \\
(\forall\, i \in 1\,..\,\#tr \bullet (\exists\, t \in \mathbb{T} \bullet \\
\quad t_s(i) \leqslant t \leqslant t_e(i+1) \wedge \\
\quad tr(i) = \mathit{event}(o(i)).t)) \wedge \\
(\forall\, e \in X \bullet (\nexists st \in S, op \in O, t, t_1, t_2 \in \mathbb{T} \bullet \\
\quad t_1 \leqslant t \leqslant t_2 \wedge e = \mathit{event}(op).t \wedge \\
\quad (s \frown \langle st \rangle, o \frown \langle op \rangle, t_s \frown \langle t_1 \rangle, t_e \frown \langle t_2 \rangle) \in R))\}
\end{aligned}$$

For Real-Time Object-Z classes to be combined using CSP operators, the set of failures F derived for a class must satisfy the following properties [12].

$$(\langle\,\rangle, \varnothing) \in F \tag{F1}$$
$$(tr_1 \frown tr_2, \varnothing) \in F \Rightarrow (tr_1, \varnothing) \in F \tag{F2}$$
$$(tr, X) \in F \wedge Y \subseteq X \Rightarrow (tr, Y) \in F \tag{F3}$$
$$(tr, X) \in F \wedge (\forall\, y \in Y \bullet (tr \frown \langle y \rangle, \varnothing) \notin F) \Rightarrow (tr, X \cup Y) \in F \tag{F4}$$

These properties hold for the definition of $failures_{rt}$ as shown below.

Proof of $F1$.
$C_{\{\vec{c} \mapsto \vec{v}\}}$ is regarded as well-defined by Smith [15] only if there exists a possible initial state of C satisfying the assignment of values to constants, $\{\vec{c} \mapsto \vec{v}\}$. Hence, if a failures semantics is given to $C_{\{\vec{c} \mapsto \vec{v}\}}$ then $\exists (s, o, t_s, t_e) \in R \bullet \{\vec{c} \mapsto \vec{v}\} \subseteq s(1)$.
By $R4$, therefore, it follows that $(\langle s(1) \rangle, \langle\,\rangle, \langle\,\rangle, \langle t \rangle) \in R$, for some $t : \mathbb{T}$. Since $\langle s(1) \rangle \in S^*$, the trace of events corresponding to the operation sequence $\langle\,\rangle$ is $\langle\,\rangle$, and $\forall\, e \in \varnothing \bullet P$ is true for any predicate P, it follows that $(\langle\,\rangle, \varnothing) \in failures_{rt}(C_{\{\vec{c} \mapsto \vec{v}\}})$. □

Proof of $F2$.
If $(tr_1 \frown tr_2, \varnothing) \in failures_{rt}(C_{\{\vec{c} \mapsto \vec{v}\}})$ then, by the definition of $failures_{rt}$, $\exists (s, o, t_s, t_e) \in R \bullet \{\vec{c} \mapsto \vec{v}\} \subseteq s(1) \wedge (\forall\, i \in 1 \,..\, \#(tr_1 \frown tr_2) \bullet (\exists\, t \in \mathbb{T} \bullet t_s(i) \leqslant t \leqslant t_e(i+1) \wedge (tr_1 \frown tr_2)(i) = event(o(i)).t))$.
If $s = s_1 \frown s_2$, $o = o_1 \frown o_2$, $t_s = t_{s1} \frown t_{s2}$ and $t_e = t_{e1} \frown t_{e2}$ such that $\#o_1 = \#tr_1 = \#t_{s1}$ and $\#s_1 = \#o1 + 1 = \#t_{e1}$ then $(s_1, o_1, t_{s1}, t_{e1}) \in R$ by $R4$.
Since $s_1(1) = s(1)$ it follows that $\{\vec{c} \mapsto \vec{v}\} \subseteq s_1(1)$. Also, $s_1 \in S^*$ and $(\forall\, i \in 1 \,..\, \#tr_1 \bullet (\exists\, t \in \mathbb{T} \bullet t_{s1}(i) \leqslant t \leqslant t_{e1}(i+1) \wedge tr_1(i) = event(o_1(i)).t))$. Hence, since $\forall\, e \in \varnothing \bullet P$ is true for any predicate P, it follows that $(tr_1, \varnothing) \in failures_{rt}(C_{\{\vec{c} \mapsto \vec{v}\}})$. □

Proof of $F3$.
Since $(\forall\, e \in X \bullet P) \Rightarrow (\forall\, e \in Y \bullet P)$ for any predicate P when $Y \subseteq X$, if $(t, X) \in failures_{rt}(C_{\{\vec{c} \mapsto \vec{v}\}})$ and $Y \subseteq X$ then $(t, Y) \in failures_{rt}(C_{\{\vec{c} \mapsto \vec{v}\}})$. □

Proof of $F4$.
If $(tr, X) \in failures_{rt}(C_{\{\vec{c} \mapsto \vec{v}\}})$ then, by the definition of $failures_{rt}$, $\exists (s, o, t_s, t_e) \in R \bullet s \in S^* \wedge \{\vec{c} \mapsto \vec{v}\} \subseteq s(1) \wedge \#tr = \#o \wedge (\forall\, i \in 1 \,..\, \#tr \bullet (\exists\, t \in \mathbb{T} \bullet t_s(i) \leqslant t \leqslant t_e(i+1) \wedge tr(i) = event(o(i).t))$.
Since $\forall\, e \in \varnothing \bullet P$ is true for any predicate P, it then follows that $(tr \frown \langle y \rangle, \varnothing) \in failures_{rt}(C_{\{\vec{c} \mapsto \vec{v}\}})$ unless $\nexists st \in S, op \in O, t, t_1, t_2 \in \mathbb{T} \bullet t_1 \leqslant t \leqslant t_2 \wedge y = event(op).t \wedge (s \frown \langle st \rangle, o \frown \langle op \rangle, t_s \frown \langle t_1 \rangle, t_e \frown \langle t_2 \rangle) \in R$.
Therefore, if $\forall\, y \in Y \bullet (tr \frown \langle y \rangle, \varnothing) \notin failures_{rt}(C_{\{\vec{c} \mapsto \vec{v}\}})$ then $(tr, X \cup Y) \in failures_{rt}(C_{\{\vec{c} \mapsto \vec{v}\}})$. □

5 Alternative Approaches Based on Object-Z

The integration of Real-Time Object-Z and CSP enables specifications of concurrent systems with real-time properties which interact with continuous variables in their environment. Having Object-Z as the basis of the integration enables complex data structures and data manipulations within the components of such systems to be constructed incrementally. Both inheritance and polymorphism are useful in this respect.

For example, suppose that the weigh units need to additionally control a gate at the end of the weighbridge to separate cows of different weight. This would be necessary, for instance, to remove older calves from their mothers. In times of drought, calves are known to drink milk from their mother for extended periods of time often to the detriment of the mother's health.

To avoid complicating the specification of the *WeighUnit* class, this aspect of the weighbridge could be added via inheritance as follows. We assume there are two gates, identified by the numbers 1 and 2, and that cows whose weights lie above $W_1 : \mathbb{R}_+$ but below $W_2 : \mathbb{R}_+$ should go through gate 1. All others should go through gate 2.

GateControlUnit

WU

$gate : \{1, 2\}$

Weigh

$\Delta(gate)$

$gate' = 1 \Leftrightarrow W_1 < (\int_{\tau}^{\tau'} weight?)/(\tau' - \tau) < W_2$

The class inherits *WU*, i.e., the class *WeighUnit* extended with a number as in Section 2, and adds a state variable *gate* denoting the gate which is currently opened. If desired, this variable could be related via a timed trace predicate to continuous output variables representing signals to the gate mechanisms. The operation *Weigh* is extended so that *gate* will be set to 1 when the recorded weight is in the range $W_1 \ldots W_2$, and set to 2 otherwise.

If we then wanted to specify a system where weighbridges may optionally be connected to gates, we could use polymorphism as follows.

$$WeighingSystem(n, w_1, \ldots, w_n) = (|||_{i=1}^{n} \downarrow WU_{\{weight? \mapsto w_i, number \mapsto i\}}) \ {}_X\|_Y \ SU$$

where $X = \{|\ Weigh, SendData\ |\}$ and $Y = \{|\ SendData, UpdateDay\ |\}$. The notation $\downarrow WU$ denotes the class *WU* or any class inherited from *WU* (in this case, the class *GateControlUnit*). The actual class of each instance is chosen nondeterministically.

Given the usefulness of these structuring concepts for larger-scale systems, we focus our discussion, in this section, on alternative approaches to specifying concurrent real-time systems which are based on Object-Z.

5.1 Other Approaches Using Real-Time Object-Z

The issue of concurrency in Real-Time Object-Z is addressed by Smith and Hayes [20]. One option explored is the use of object instantiation (as in standard Object-Z). Given an object a of a class C (declared as $a : C$), we can refer to a variable or constant x of the object by the notation $a.x$. Similarly, we can state that the object satisfies its class's initial condition or undergoes operation Op of its class by $a.\mathit{INIT}$ and $a.Op$ respectively. Adopting this approach, we might specify the weighing system as follows.

WeighingSystem

$$wu : \mathbb{P}\, WU$$
$$su : SU$$
$$\forall a, b : wu \bullet a.number = b.number \Rightarrow a = b$$

INIT

$$\forall a : wu \bullet a.\mathit{INIT}$$
$$su.\mathit{INIT}$$

$$Weigh == [a : wu] \bullet a.Weigh$$
$$SendData == [a : wu] \bullet a.SendData \parallel su.RecData$$
$$UpdateDay \mathrel{\widehat{=}} su.UpdateDay$$

This class comprises a set of objects of class WU, each with a unique number, and an object of class SU. We assume that the real-time properties of the objects' classes are implicitly maintained. The initial condition and operations of the class are constructed from those of the component classes using Object-Z operation operators [16, Chapter 3].

Synchronisation, including communication, is explicitly modelled using the $\parallel$ operator. This operator conjoins its argument operations and equates any inputs in one with outputs in the other when they have common basenames. Similarly, concurrent occurrences of operations (whether as part of synchronisation or not) are explicitly specified via the conjunction of operations.

As pointed out by Smith and Hayes, however, this approach is undesirable for two reasons.

- Explicitly stating all combinations of operations which can occur concurrently may become unwieldy for large systems comprising many components.
- Conjoined operations are forced to have the same start and end times. Hence, the partial overlap of operations cannot be specified.

An alternative approach explored by Smith and Hayes is the definition of a parallel composition operator for combining classes. This operator is similar to the CSP operator in that it ensures the synchronisation of common-named operations. However, any synchronising operations must still have the same start and end times.

A more fundamental problem with this approach is that the single operator introduced hinders direct specification of many systems. For example, the weighing system specification in this paper makes use of CSP's interleaving operator |||. Using parallel composition alone, in this case, would require *Weigh* and *SendData* operations to include the weigh unit's number as a parameter (to avoid weigh units synchronising on these operations).

The approach could be extended with an interleaving operator, as well as other operators such as renaming and hiding found in CSP. However, the result would not improve on an integration with CSP itself as detailed in this paper. Furthermore, the introduction of new notations, as opposed to the integration of existing notations, result in an approach which is both less familiar and less amenable to use with existing tools and verification and refinement techniques.

5.2 Integrating Object-Z/CSP with the Timed Trace Notation

An alternative to integrating Real-Time Object-Z with CSP would be to integrate Object-Z/CSP with the timed trace notation giving a timed trace semantics to specifications. In such an approach, timed trace predicates could be used to restrict the time of occurrence of events within an Object-Z/CSP system specification.

For example, given the Object-Z/CSP specification of the weighing system in Section 2, the following effect predicates might be added to ensure the *UpdateDay* event occurs as required.

$$\langle \delta = 24 * \text{hour} \rangle \subseteq \langle true \rangle \,;\, \langle \mathit{UpdateDay} \rangle \,;\, \langle true \rangle$$
$$\langle \mathit{UpdateDay} \rangle \,;\, \langle \neg\ \mathit{UpdateDay} \rangle \,;\, \langle \mathit{UpdateDay} \rangle \subseteq \langle \delta = 24 * \mathit{hours} \rangle$$

The first predicate (identical to that in the Real-Time Object-Z specification in Section 3) ensures that *UpdateDay* occurs at least once in every 24 hours. The second predicate ensures that the separation between *UpdateDay* events is precisely 24 hours. (Note that the intervals $\langle \mathit{UpdateDay} \rangle$ here correspond to event occurrences and hence comprise a single point of time only.)

One problem with this approach is that because we are constraining CSP processes, rather than Object-Z classes, we can only specify timing constraints on operation occurrences. However, it is also often desirable to specify timing constraints on state variables as evidenced by the case studies by Smith and Hayes [19,20].

A more serious problem becomes obvious if we try to specify the timing constraints on the events *Weigh* and *SendData*. These constraints need to be specified for each weigh unit and not just for the system as a whole. For example, we need to ensure that *each* weigh unit performs *SendData* every 10 minutes. This can only be done if we can distinguish *Weigh* events from individual weigh units at the system level. This could be done only by adding their number, or some other unique identifier, as a parameter to the event.

In general, however, we cannot, in this approach, add timing constraints to components of a specification before composing them. This would give the components a timed trace semantics and hence invalidate the use of standard

CSP operators. Therefore, the approach is not compositional in the sense that a system specification, together with timing constraints, could not be used as a component in the specification of another system. This limits the use of such an approach for specifying larger-scale systems.

5.3 Integrating Object-Z with Timed CSP

A final alternative is to integrate Object-Z with Timed CSP [1]. Timed CSP extends CSP with operators for modelling delays, timeouts and timed interrupts. The approach in this case would be to associate a Timed CSP process definition with each Object-Z class in order to control the timing of events associated with operations. Such an approach has been adopted in RT-Z [22], an integration of Z and Timed CSP, and TCOZ [11], an integration of Object-Z and Timed CSP.

Since CSP (and, hence, Timed CSP) processes are conceptually driven by their environment, the timing operators are limited in order that the maximum or exact time between events cannot be specified. Otherwise, a process could restrict its environment. Therefore, the timing properties on the operations *Weigh* and *UpdateDay* of the classes *WeighUnit* and *StoreUnit*, respectively, cannot be readily captured with this approach.

They can be captured, however, if we can guarantee the event in question is not refused by the process's environment. One way of doing this is to make the event internal to the process by hiding it. For example, the constraint on *StoreUnit* is captured by the following process.

$$
\begin{aligned}
SU_{Behav} = {} & (\mu\, R \bullet RecData?data \rightarrow R) \\
& ||| \\
& (\mu\, U \bullet Wait\ 24 * \text{hour};\ UpdateDay \rightarrow U) \\
& \setminus \{UpdateDay\}
\end{aligned}
$$

This process allows *RecData* events to occur as often as necessary, and each *UpdateDay* event to occur only after a delay of 24 hours. Since the latter event is internal and cannot be refused by the environment, it occurs immediately after the 24 hour delay.

This approach is not suitable for the timing constraint on *SendData* of class *WeighUnit* however. This event cannot be hidden since it needs to synchronise with the corresponding event of *StoreUnit*. To specify this constraint, we could complement the process description with explicit predicates on the (timed) failures of the process. Alternatively, we could place a constraint on the environment of *WeighUnit* stating that it never refuses a *SendData* event. The use of such environmental constraints is discussed by Schneider [13].

Another approach is to introduce additional operators, such as the "deadline" command of TCOZ [10], to model the fact that events must occur before or at particular times. The use of such additional operators, however, goes against our desire to integrate existing notations for reasons of familiarity and reuse of existing tools and techniques. In particular, the addition of a dealine command changes the specification paradigm of Timed CSP to one in which processes are able to "force" synchronising events in their environment to occur. This makes

it unlikely that methods of compositional refinement developed for Timed CSP would be applicable. In contrast, our approach uses untimed CSP and time is represented simply as an event parameter. Hence, CSP refinement methods are still applicable.

Timed CSP also differs from the approach of this paper in that it does not support modelling constraints on continuous variables. In addition, the style of specification is fundamentally different. Timing constraints are modelled in a very operational fashion via delays, timeouts and interrupts. This can sometimes be cumbersome. In contrast, using the timed trace notation of Real-Time Object-Z, timing constraints can be modelled declaratively as predicates on time intervals. The approach is more abstract resulting in more concise specifications.

6 Conclusion

In this paper, we have brought together two tracks of research on integrating formal methods (Object-Z/CSP and Real-Time Object-Z) and, in doing so, three modelling paradigms: state-based (Object-Z), event-based (CSP) and trace-based (the timed trace notation). The result is a specification notation that is capable of modelling complex data structures, concurrency, real-time constraints and continuous variables.

By developing a semantic integration where the individual notations are separated in specifications, we have aimed at increasing the accessibility of the approach to specifiers already familiar with one or more of the notations, and the amenability of using existing tools and verification and refinement techniques with the approach. The latter is an area of future work.

Acknowledgements

Thanks to Ian Hayes and Kirsten Winter for comments on an earlier draft of this paper. Thanks also to Jim Davies, Steve Schneider and Carsten Sühl for improving my understanding of Timed CSP. This work is funded by Australian Research Council Large Grant A49801500, *A Unified Formalism for Concurrent Real-Time Software Development*.

References

1. J. Davies and S. Schneider. A brief history of Timed CSP. *Theoretical Computer Science*, 138(2):243–271, 1995.
2. J. Derrick and G. Smith. Structural refinement in Object-Z/CSP. In W. Grieskamp, T. Santen, and B. Stoddart, editors, *2nd International Conference on Integrated Formal Methods (IFM'00)*, volume 1945 of *Lecture Notes in Computer Science*, pages 194–213. Springer-Verlag, 2000.
3. R. Duke and G. Rose. *Formal Object-Oriented Specification using Object-Z.* MacMillan, 2000.
4. C.J. Fidge, I.J. Hayes, and B.P. Mahony. Defining differentiation and integration in Z. In J. Staples, M.G. Hinchey, and Shaoying Liu, editors, *IEEE International Conference on Formal Engineering Methods (ICFEM '98)*, pages 64–73. IEEE Computer Society Press, 1998.

5. C.J. Fidge, I.J. Hayes, A.P. Martin, and A.K. Wabenhorst. A set-theoretic model for real-time specification and reasoning. In J. Jeuring, editor, *Mathematics of Program Construction (MPC'98)*, volume 1422 of *Lecture Notes in Computer Science*, pages 188–206. Springer-Verlag, 1998.
6. C. Fischer. How to combine Z with a process algebra. In J.P. Bowen, A. Fett, and M.G. Hinchey, editors, *11th International Conference of Z Users*, volume 1493 of *Lecture Notes in Computer Science*, pages 5–23. Springer-Verlag, 1998.
7. C. Fischer and G. Smith. Combining CSP and Object-Z: Finite or infinite trace semantics? In T. Higashino and A. Togashi, editors, *Formal Description Techniques and Protocol Specification, Testing, and Verification (FORTE/PSTV '97)*, pages 503–518. Chapman and Hall, 1997.
8. C. Fischer and H. Wehrheim. Model-checking CSP-OZ specifications with FDR. In K. Araki, A. Galloway, and K. Taguchi, editors, *1st International Conference on Integrated Formal Methods*, pages 315–334. Springer-Verlag, 1999.
9. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
10. B.P. Mahony and J.S. Dong. Sensors and actuators in TCOZ. In J. Wing, J.C.P. Woodcock, and J. Davies, editors, *World Congress on Formal Methods (FM'99)*, volume 1709 of *Lecture Notes in Computer Science*, pages 1166–1185. Springer-Verlag, 1999.
11. B.P. Mahony and J.S. Dong. Timed Communicating Object Z. *IEEE Transactions on Software Engineering*, 26(2):150–177, 2000.
12. A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.
13. S. Schneider. *Concurrent and Real-Time Systems: The CSP Approach*. John Wiley & Sons, 1999.
14. G. Smith. A fully abstract semantics of classes for Object-Z. *Formal Aspects of Computing*, 7(3):289–313, 1995.
15. G. Smith. A semantic integration of Object-Z and CSP for the specification of concurrent systems. In J. Fitzgerald, C.B. Jones, and P. Lucas, editors, *Formal Methods Europe (FME'97)*, volume 1313 of *Lecture Notes in Computer Science*, pages 62–81. Springer-Verlag, 1997.
16. G. Smith. *The Object-Z Specification Language*. Advances in Formal Methods. Kluwer Academic Publishers, 2000.
17. G. Smith and J. Derrick. Refinement and verification of concurrent systems specified in Object-Z and CSP. In M.G. Hinchey and Shaoying Lui, editors, *First International Conference on Formal Engineering Methods (ICFEM '97)*, pages 293–302. IEEE Computer Society Press, 1997.
18. G. Smith and J. Derrick. Specification, refinement and verification of concurrent systems – an integration of Object-Z and CSP. *Formal Methods in System Design*, 18(3):249–284, 2000.
19. G. Smith and I.J. Hayes. Towards real-time Object-Z. In K. Araki, A. Galloway, and K. Taguchi, editors, *1st International Conference on Integrated Formal Methods (IFM'99)*, pages 49–65. Springer-Verlag, 1999.
20. G. Smith and I.J. Hayes. Structuring Real-Time Object-Z specifications. In W. Grieskamp, T. Santen, and B. Stoddart, editors, *2nd International Conference on Integrated Formal Methods (IFM'00)*, volume 1945 of *Lecture Notes in Computer Science*, pages 97–115. Springer-Verlag, 2000.
21. J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 2nd edition, 1992.
22. C. Sühl. RT-Z: An integration of Z and timed CSP. In K. Araki, A. Galloway, and K. Taguchi, editors, *1st International Conference on Integrated Formal Methods (IFM'99)*, pages 29–48. Springer-Verlag, 1999.