

The Design of a Tool-Supported Graphical Notation for Timed CSP

Phillip J. Brooke¹ * and Richard F. Paige²

¹ School of Computing, University of Plymouth, Drake Circus, Plymouth,
Devon, PL4 8AA, U.K.

philb@soc.plym.ac.uk

² Department of Computer Science, University of York, Heslington, York, YO10 5DD, U.K.

paige@cs.york.ac.uk

Abstract. A graphical notation for representing Timed CSP (TCSP) specifications is presented. The notation, which integrates features from a number of existing specification languages, including Statecharts, is aimed at providing the means for more easily constructing and managing large TCSP specifications, with the intention of forming the basis for tools and a methodology for applying TCSP in the large. The graphical notation extends TCSP by allowing specifications to be both processes and arbitrary predicates, thus increasing the expressiveness and applicability of the notation. An extendible tool framework, designed for the graphical notation and to be integrated with other tools, is presented. We discuss the features of this framework, especially how it aims to support reasoning about TCSP specifications.

Keywords: Timed CSP, process algebra, graphical notation, tool support.

1 Introduction

The general theme of this paper is the usability of formal specification languages, particularly as they apply to large-scale software development. Our aim is to improve the usability of one formal specification language – Timed CSP – in this domain, in part by integrating techniques that have proven successful, and by providing an extendible basis for tool support for reasoning and validation of specifications.

Timed CSP (TCSP) [6] has been proposed as a specification language for modelling and reasoning about concurrent, communicating systems that must obey explicit timing constraints. The notation of TCSP is based on the process algebra CSP [13]. The syntax of TCSP and related notations may be unpopular with system engineers, especially those with little experience with formal specification languages and mathematical notations in general. Moreover, TCSP is generally difficult to use in modelling and describing large systems, in part because of its limited support for a notion of module, and also because of its design: it is intended to capture the behaviour of small, yet critical components of software systems, and to support reasoning about them. As well, limited methodological support exists for TCSP; this arises typically through use of tools such

* Corresponding author.

as FDR [8] (although issues arise with using these tools when working in the timed domain [3]). Methodological support is vital if languages and tools are to be usable for development in the large.

In this paper, we propose a graphical notation for Timed CSP, designed to make TCSP more appealing to system engineers, to make the notation more appropriate and usable for large-scale modelling, and to provide a foundation for further tool support. Tool support will be particularly vital for drawing models and for generating textual representations of TCSP specifications (and encodings of TCSP specifications in other languages, such as PVS [26] or FDR). The graphical notation is constructed by integrating proven techniques from other notations, particularly textual TCSP and Harel's Statecharts [12].

We discuss design criteria for the graphical notation in Section 2. To give an overview, the notation has a formal syntax and semantics, is designed to be simple, usable, scalable, and, we claim, intuitive. Our intention is that this notation will appeal to systems engineers with less experience of formal methods, while providing the necessary formality of syntax and semantics that can be exploited with automated tools such as FDR and PVS. At the same time, we do not desire to introduce a graphical notation that will require developers experienced with textual TCSP to overcome a substantial learning curve to deal with in order to work with the new notation.

The notation will also extend the expressive power and applicability of TCSP by allowing specifications to be both processes and predicates. This is particularly desirable for modelling in the large, in part because it allows, in certain cases, simpler specifications to be written, and also because it makes it easier for TCSP to be integrated with other formal specification languages and their tools.

1.1 Overview

We start by briefly examining the desirable characteristics of our graphical notation – and similar notations in general – based on work in [3, 18]. We also discuss the related work upon which our notation and its supporting tools are founded. Then we summarise the syntax for Timed CSP. (In this paper, we do not concern ourselves with the semantics of TCSP, but only with the meaning of the notations we describe. Effectively, the semantics of graphical TCSP will be given in terms of TCSP programs.) From there, we propose our graphical notation, give an example, and explain the semantics of the graphical notation, which is constructed via a mapping from the grammar of the graphical notation to its semantic domain. The syntax and semantics of the graphical notation are not described via a metamodel, as is currently common in the object-oriented realm [5, 19]; rather the syntax is captured via integrating context-free grammars with graphical terms, and by defining a precise mapping from sentences generated by the grammar to sentences in textual TCSP.

We explain the extension of the notation to predicate specifications, and the benefits obtained from doing this. We then outline an extendible tool framework for supporting the graphical notation. The framework is designed to be integrated with a variety of additional tools, including pretty-printers, documentation generators, model checkers, and theorem provers. We describe how this tool integration works, and we end with a discussion of ongoing and future work, and some conclusions.

2 Background and Design Goals

We give a short overview of related work, and particularly discuss design criteria for the graphical notation.

2.1 Related Work

There has been much related work on graphical notations for modelling and building software systems. These languages have been used in particular for constructing new systems (e.g. to capture requirements or architectural designs), and for describing existing systems (e.g. via reverse-engineered models such as UML diagrams or architectural interconnection diagrams). Our particular focus in this paper is on the former class of languages, particularly those suitable for formal reasoning. As well, we are particularly focused on the development of graphical notations for existing textual notations.

Statecharts [12] is a formal language for modelling real-time and reactive systems. It is a tool-supported graphical language based on finite state machines. It possesses extra notation for managing and packaging large specifications, and for capturing timing constraints and triggers for allowing transitions to be taken. Statecharts possesses a formal semantics and is supported by the Statemate tool, which can be used for simulation, code generation, and automated reasoning. A number of formal semantics have been proposed for Statecharts, including the official semantics provided by the Statemate tool.

The Unified Modelling Language (UML) [2] is a language for visualising systems of all kinds from a variety of different perspectives, including static structure, behaviour, and user interactions. The UML is not a formal language; it does not possess a formal semantics. A variety of tools are available for producing UML specifications, and for validating them against well-formedness constraints. Tools such as Rational Rose and Together/J are particularly useful for visualising programs, e.g. in Java or C++. UML profiles [5] can be used for defining specific dialects of UML that are more appropriate for visualising programs in specific languages.

The work of Baresi and Pezzè et al. [1] has focused on providing a formal semantics for Structured Analysis diagrams, e.g. data flow diagrams. Their work showed that the numerous different interpretations of data flow diagrams could be carefully structured and classified, so that the diagrams should actually be considered a family of languages. In other words, data flow diagrams are semantically fragmented, and formalisation of such diagrams involves selecting a member of the family of interest and using a specially crafted formalisation appropriate for that family member. This discovery provides further motivation for UML profiles, and is also the same problem that has arisen with Statecharts. It also illustrates some of the complications with providing a formal semantics to a notation after the notation has been put into widespread use.

Dong et al. [7] have recently considered visualising Timed Communicating Object-Z specifications on the web, via use of XML/XMI, and by using UML diagrams as projected views. This work is similar in intent to our own: the construction of lightweight tools to support formal methods. This project emphasises visualisation, as well as the use of Z and Object-Z. While we also emphasise visualisation, we focus strictly on

TCSP, and we also desire to support automated reasoning via tool integration; reasoning is not yet considered in Dong's work [7]. However, Dong's use of XML as an infrastructure suggests that it should be possible for their toolset to be extended to support interfacing with a variety of reasoning tools. We also do not require nor desire the use of UML diagrams for visualisation, as we do not need object-oriented modelling features for capturing TCSP. We also desire to avoid the semantic problems associated with UML, such as its imprecise semantics for state machines.

There is an increasing amount of work on the subject of *visualisation* [9, 10, 20] of information, programs, and systems. One general conclusion from this work is that the utility of a notation is entirely dependent on context. A further inference is that for reasoning and programming, textual notations are invariably preferable (since for this type of work, details are relevant and needed), whereas for other tasks such as architectural design, analysis, and explanation of systems, graphical notations are often preferable.

2.2 Design Goals

Timed CSP, being textually based, is not entirely appropriate for describing large-scale systems. While it does provide support for abstraction – a necessary mechanism in modelling languages – it does not provide large-scale structuring mechanisms such as modules, class, components, or packages. Graphical notations are generally considered to be suitable, though certainly not sufficient, for large-scale modelling and development, in part because of their capabilities at abstraction, because of their readability and usability, and because of their ability of being able to present many critical architectural aspects of systems succinctly and quickly.

Textual notations are essentially one dimensional. Indenting of programs can make their interpretation easier. Similarly, formulæ can be structured in a manner to assist the reader, as in Lamport's work [15], or using the structured calculational proof style [11]. On the other hand, graphical notations have two dimensions, and there are several ways in which the notations can be structured. Two common types of structuring are graphs or trees and nested boxes. In each case, the nodes, arcs or boxes can be decorated with textual symbols, dashes, arrows, etc. We can also combine both mechanisms of structuring, which is what is done in Statecharts, and also in several of the diagrams of UML, e.g., packages.

A desirable graphical notation will not use colours and subtly distinct decorations or annotations, since they may be difficult to print (in the case of monochrome printers) in the former case, or difficult to hand-draw in the latter case. Similarly, we rule out the use of fonts and font styles on the grounds that they are difficult for humans to draw. For example, UML allows use of italic font to indicate that classes are abstract (i.e. they cannot be instantiated). This is clearly useful only when tools are available; for hand-drawn models, textual annotations are preferable (and are, in fact, what UML users typically apply in this particular case).

In general, specifications written using a graphical language will take up more space than specifications written using a similar textual language. It is undesirable for a graphical notation to be explosive, in that its specifications tend to grow rapidly in size (mea-

sured in terms of amount of space taken on the screen or on a piece of paper) as further elements are added to the specification.

We have the following specific design goals for our graphical notation for TCSP. For a detailed overview of general design goals for modelling language, the reader is referred to [18], where principles for designing languages are discussed. Leveson et al. have also described desirable design criteria for graphical languages [16]. The following list of requirements is also partially based on these criteria.

1. *Usability in the large.* It should be possible to structure and manage large TCSP specifications. This ability will arise by being able to omit details of a specification where necessary, and focus on presentation of architectural details. For TCSP, this means being able to focus on selected process connectors, e.g. nondeterministic choice, parallel composition, etc. This is analogous to modelling languages like UML where it is possible to describe the abstractions and connectors in a system without describing the details of the abstractions at the same time.
2. *Simplicity.* The notation should be simple to draw, both by hand and by automated tool. It should require use of a minimum number of marks and graphical cues in order to distinguish between the different TCSP elements graphically. It should provide a unique way of describing each TCSP construct, following the *principle of uniqueness* described by Meyer [17].
3. *Tool-supportable.* The graphical syntax should be defined so as to be easy to support by automated tools. In particular, it is desirable to have a GUI-based tool for constructing the graphical TCSP specifications. As well, it is desirable to be able to use an assortment of automated tools, such as model checkers (e.g. FDR) and theorem provers (e.g. PVS) for reasoning about the specifications. At the same time, it is difficult to predict the exact types of different reasoning that will be need on TCSP specifications, and thus it is desirable to have an extendible *tool framework* in which different reasoning tools can be integrated or removed as the need arises. This is discussed further in the sequel.
4. *Suitable for use without tools.* Though the notation must be supportable by tools, we also desire that it be easy to hand-draw, since many developers will want to use a graphical language as a rough-sketch tool.
5. *Clear relationship with textual TCSP.* The relationship between elements of the graphical TCSP and the textual TCSP should be obvious, so as to reduce the learning curve for the new syntax, to improve *traceability*, and to define a clear and precise mapping between a graphical TCSP specification and its formal semantics.
6. *Seamlessness and reversibility.* A graphical TCSP specification will eventually be transformed into textual TCSP, so as to make use of tools and to produce executable code. The mapping from graphical TCSP to textual TCSP should be seamless [17]: the same abstractions should be used in each syntax, and a precise mapping should be defined between them. The mapping should also be reversible, so that it should be possible to automatically produce graphical TCSP specifications from textual TCSP specifications, thus enabling round-trip engineering [17].
7. *Semantically consistent.* A diagram should be interpreted in exactly and only one way. This is essential for a language that is to be used for formal specification and reasoning.

The issue of traceability, in point 5 above, is a critical one, especially when considered in parallel with point 4. The concept of a tool framework discussed in point 3 suggests a graphical front-end with which different back-end tools can be integrated, e.g. FDR. Back-end reasoning tools will in general require text-based input, and will produce text-based feedback. It should be possible to easily map concepts in the graphical notation to concepts in textual TCSP - and vice versa - so that feedback from tools can be more easily integrated into specifications.

Point 1 above discusses usability, particularly in the large. Whether or not a particular notation is usable or aesthetically pleasing is a subjective matter. Graphical notations are generally considered more appropriate for capturing large amounts of information, and for presenting it in such a way so that details can be ignored where needed. However, graphical notations, by virtue of using their layout for imparting information, rather than textual symbols, suffer more from *secondary notation*, such as layout and typographic cues, than textual notations [21]. It is this secondary notation that sometimes appeals to users, and causes confusion in interpreting such notations. We aim to avoid use of secondary notation in the design of our graphical notation.

3 Overview of Timed CSP

We will now describe the textual notation of Timed CSP. Timed CSP has a simple grammar, as follows (P_T represents a sentence in Timed CSP).

$$\begin{aligned}
P_T ::= & \text{Stop} \mid \text{Skip} \mid \text{Wait } t \mid a \rightarrow P_T \mid P_T; P_T \mid P_T \square P_T \mid P_T \sqcap P_T \\
& \mid a : A \rightarrow P_{T_a} \mid P_T \triangleright_t P_T \mid P_T \triangle P_T \mid P_T \otimes_t P_T \mid f(P_T) \mid l : P_T \\
& \mid P_T \setminus A \mid \parallel_A P_T \mid P_{TA} \parallel_A P_T \mid P_T \parallel P_T \mid P_T \parallel_A P_T \mid \mu X \bullet F(X)
\end{aligned}$$

A description of the meaning of each operator and terminal now follows:

- Stop* This will never engage in any external communication; it is the broken program.
- Skip* Do nothing except terminate (event ✓); it is ready to terminate immediately.
- Wait t* This does nothing, but is ready to terminate after delay t .
- $a \rightarrow P$ This program is initially prepared to engage in event a ; it then behaves as P . It is right associative, i.e., $a \rightarrow b \rightarrow P \equiv a \rightarrow (b \rightarrow P)$.
- $P; Q$ Sequential composition transfers control from P to Q when P performs ✓. Right associative.
- $P \square Q$ The environment is offered a choice between the programs P and Q . If the environment will cooperate with both, then the choice is nondeterministic. Right associative.
- $P \sqcap Q$ The outcome between P and Q is a nondeterministic (internal) choice. Right associative.
- $a : A \rightarrow P_a$ This program offers an external choice of events drawn from the (possibly infinite) set A .

We can use this to build the channel notation: for a channel c carrying values of type T and $P'_{c.v} \equiv P_v$

$$- c?x : T \rightarrow P_x \equiv a : \{v : T \mid c.v\} \rightarrow P'_a$$

– $c!v \equiv c.v$

$P \triangleright_t Q$ (Timeout.) After delay t , if no communications have occurred, then control is passed to Q . If the first communication with P occurs at exactly t , then the result is nondeterministic. This is internal choice in the untimed model.

$P \Delta Q$ (Interrupt.) This behaves as P until the first event of Q , when control passes to Q .

$P \circlearrowleft_t Q$ (Transfer.) Control passes from P to Q after the delay t has passed. The first program is interrupted at some nondeterministic point in the untimed model.

$f(P)$ Observable events in P are renamed according to the function f .

$l : P$ This program behaves as P , except all events a in P are renamed $l.a$.

$P \setminus A$ This program behaves as P , except that events in A are hidden. Hidden events no longer require the cooperation of the environment, and occur as soon as P is ready to perform them.

$\parallel_{A_i} P_i$ Network parallel, where the program P_i has interface A_i : every pair of programs must cooperate on the intersection of their interface sets.

$P_A \parallel_B Q$ (Simple binary parallel.) P may only perform events in A ; Q may only perform events in B . They must agree on events in $A \cap B$.

$P \parallel \parallel Q$ (Interleave.) Both P and Q evolve concurrently without interacting, except that they must both agree on termination. Right associative.

$P \parallel_C Q$ (Hybrid parallel.) P and Q must synchronise on the events in C and agree on termination; they interleave on other events. Right associative.

$\mu X \bullet P(X)$ This defines X to be P , possibly recursively. For a well-defined semantics, P must be guarded:

- untimed model: each free occurrence of X in P must be preceded by at least one observable event
- timed model: each free occurrence of X in P must be preceded by a non-zero time delay

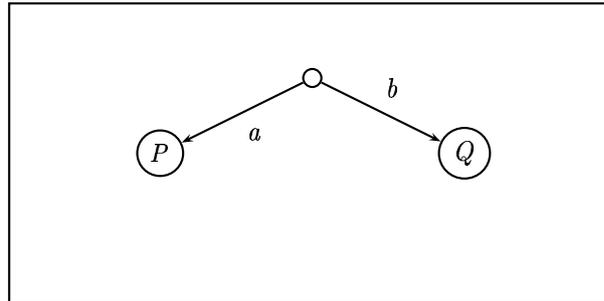
Timed CSP has a well-understood semantics [6]. We will not repeat them here.

4 Graphical Timed CSP

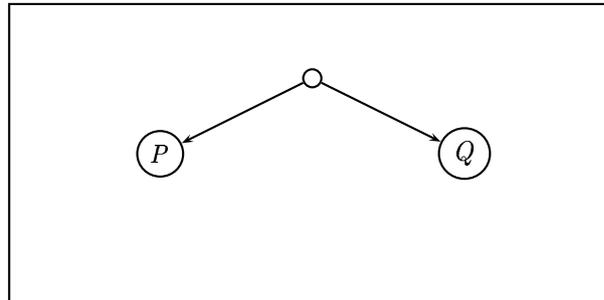
4.1 Initial Discussion of the Design

The graphical notation that we introduce in Section 4.3 uses a syntax similar, but not identical to the textual syntax of TCSP. Indeed, initially, we attempted to construct a graphical notation with the same grammar structure as for the textual notation. Hence the graphical notation would be interchangeable with the textual notation and vice versa, and the semantics of the graphical notation would be easily determined via the textual notation.

Using the graphical notation with the identical syntax, we soon ran into problems: the diagrams generated by such a syntax are not intuitive. For example, external choice does not run directly onto the prefix statements. More concretely, we might like the program $a \rightarrow P \square b \rightarrow Q$ to be rendered something like



But then, how should we treat $P \square Q$? Using the same syntax structure, we would have



This is not intuitive; there are no labels on the node, but for the program to progress, the environment must choose the leading event of either P or Q .

After this failed attempt at mimicking the grammar of the textual syntax for the graphical notation, we had a number of options to consider.

1. Conclude that graphical representations of textual notations are undesirable, and should never be used as replacements. However, work on reverse engineering and program understanding [24] suggests that this is not an appropriate conclusion.
2. Design a restricted graphical syntax. For example, in the case illustrated above, the operands of an external choice operator would be restricted to being programs which start with a prefix. This is the approach that we adopt in the next section. Moreover, we soon concluded that it would be useful if textual grammar terms and predicates could be 'plugged in' to a graphical specification. Doing this increases the expressiveness of the language, improves extendibility – particularly by making it easier to define linkages to other language and tools – and does not increase the complexity of using the language.
3. Some users will want to draw 'pretty pictures' in a tool which will thereafter generate code. Such users often do not understand the semantics of the graphical notation. Thus, a third option is to educate the users so that they understand the underlying assumptions of the language's semantics further.

Generally, whichever approach is adopted, the users need to understand what they're designing. Of course, it may then be the case that once the users are familiar with the language (graphical or textual), that they may then prefer the textual notation – it is

often quicker to write, and easier to manipulate during reasoning. Obviously, we chose approach 2; elements of approach 3 are also desirable. Based on this, we now propose our notation.

4.2 Proposal for a Graphical Notation for TCSP

We now propose our graphical notation for Timed CSP. We desire to satisfy the requirements discussed in Section 2. Typically, when the syntax of a graphical notation is presented, the syntax is described informally by example (and by cataloguing the available constructs), or by providing a metamodel, a list of rules that valid specifications written in the language must obey.

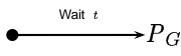
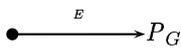
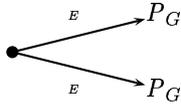
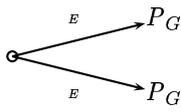
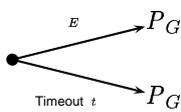
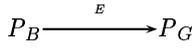
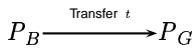
Instead of taking this approach, we present the syntax of graphical TCSP using a context-free grammar, where terminal symbols in the grammar are graphical constructs. This is useful for several reasons: the presentation of the grammar is formal, and can be implemented more straightforwardly using tools than an informal presentation; and the grammar allows substitution of equivalent textual constructs where graphical constructs are expected (thus permitting an integration of textual TCSP and graphical TCSP). Moreover, it simplifies traceability: the structure of the grammar for graphical TCSP mimics much of the structure of the grammar for textual TCSP, and as such it is straightforward to see the textual equivalent of a graphical TCSP specification, and vice versa. The metamodelling approach to defining the syntax and semantics of a language is appropriate for a complex family of languages like UML, where different views of the same system are constructed and need to be verified, but for a relatively simple language like TCSP, it is excessive.

4.3 Grammar for Graphical TCSP

We present the context-free grammar for graphical TCSP. In the grammar, P_G refers to a term in the graphical language, and P_T is a term in the textual language. We further add P_P as a process defined by a (textual) predicate; we will return to this point later. It is the mechanism by which predicate specifications can be integrated into TCSP. This is desirable from the point of view of extendibility, expressiveness, and with an eye towards future tool integration.

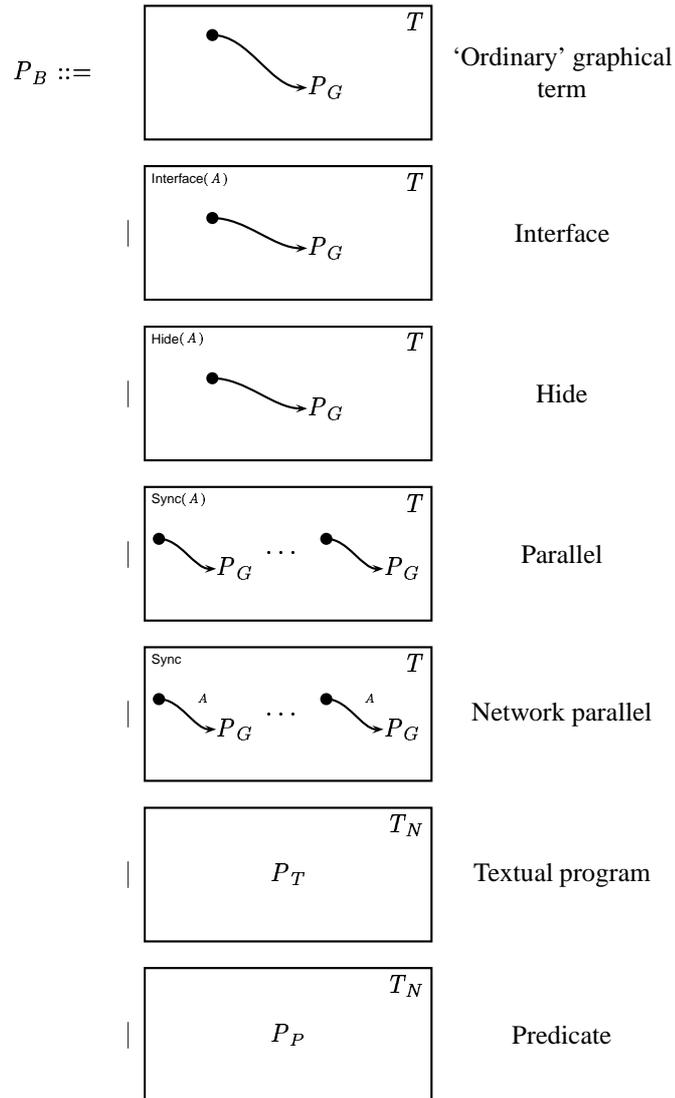
We first define terms in the graphical language. Each graphical term is annotated with a comment in the right-most column, indicating the textual TCSP term that it depicts.

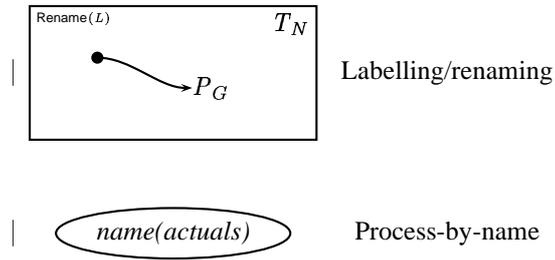
$P_G ::=$		Stop
		Skip

		Wait
		Prefix
		Sequential composition
		External choice
		Internal choice
		Timeout
		Interrupt
		Transfer
	P_B	‘Boxed’ graphical term

Although not explicitly stated in the grammar, the external and internal choices can both be generalised from 2-way to n -way (the prototype tool support described in Section 6 includes this concept (i.e. that multiple arcs leaving a node can be included). Note that the choice must be entirely external or entirely internal; it cannot be a mixture.

The last production above refers to a ‘boxed’ graphical term. We use *boxes* as our modularisation construct. They introduce scoping, and optionally, naming. P_B representing boxed graphical terms, is as follows.





The last section of the grammar fills in all the remaining details, by adding annotations for boxes (i.e., names, local variables, parameters), as well as events and renaming functions. For now, we do not specify the details of the renaming function. We envisage it working in the same way as FDR's renaming function [8].

$T ::=$	$[T_N][T_L][T_P]$	Annotations
---------	-------------------	-------------

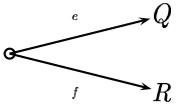
$T_N ::=$	$name$	Naming
-----------	--------	--------

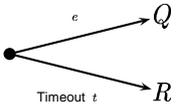
$T_N ::=$	$Local(locals)$	Local variables
-----------	-----------------	-----------------

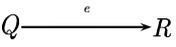
$T_P ::=$	$Params(parameters)$	Parameters
-----------	----------------------	------------

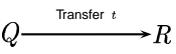
$E ::=$	$c/e : a$	condition-event- action
---------	-----------	----------------------------

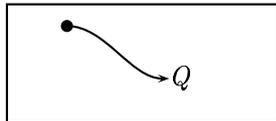
	e	(simple) event
--	-----	----------------

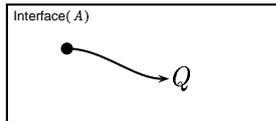
If $P =$  then $\mathcal{M}(P) = (e \rightarrow \mathcal{M}(Q)) \sqcap (f \rightarrow \mathcal{M}(R))$

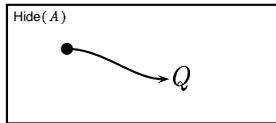
If $P =$  then $\mathcal{M}(P) = (e \rightarrow \mathcal{M}(Q)) \triangleright_t \mathcal{M}(R)$

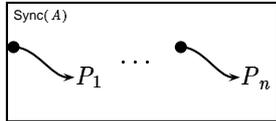
If $P =$  then $\mathcal{M}(P) = \mathcal{M}(Q) \Delta (e \rightarrow \mathcal{M}(R))$

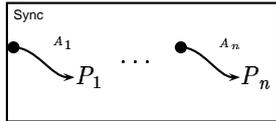
If $P =$  then $\mathcal{M}(P) = \mathcal{M}(Q) \otimes_t \mathcal{M}(R)$

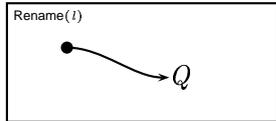
If $P =$  then $\mathcal{M}(P) = (\mathcal{M}(Q))$

If $P =$  then $\mathcal{M}(P) = (\mathcal{M}(Q) \setminus (\alpha(Q) \setminus A))$

If $P =$  then $\mathcal{M}(P) = (\mathcal{M}(Q) \setminus A)$

If $P =$  then $\mathcal{M}(P) = (\parallel_{A}^{i=1, \dots, n} \mathcal{M}(P_i))$

If $P =$  then $\mathcal{M}(P) = (\parallel_{A_i}^{i=1, \dots, n} \mathcal{M}(P_i))$

If $P =$  then $\mathcal{M}(P) = (l : \mathcal{M}(Q))$

Textual programs are included easily. Since the graphical grammar already includes productions from nonterminal P_T , they can be included in graphical specifications, and their semantics is produced simply by defining \mathcal{M} on textual terms to be the identity map.

If $P =$ $Q : P_T$ then $\mathcal{M}(P) = (Q)$

The graphical grammar includes general predicate specifications. Their semantics is included by treating them, informally, as the most general program that satisfies the predicate. More precisely, predicates are defined on a set of observations: depending on the semantic domain, this may be, for example untimed traces; timed traces; untimed failures; or timed failures.

A predicate S is defined as a function with domain equal to the semantic domain, and a range of $\{\text{true}, \text{false}\}$. Given an observation of any process, it either satisfies the predicate or it does not. The textual program we use has to be a program, R , such that R generates all observations accepted by S while not generating any observations that are not accepted by S . (For now, we ignore the possibility of unsatisfiable predicates: we would envisage some part of the tool support reporting that it cannot satisfy this predicate.)

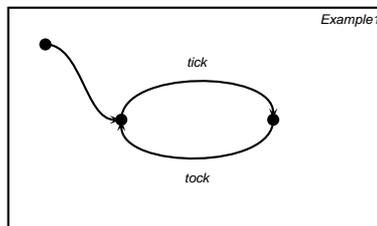
If $P =$ $S : P_P$ then $\mathcal{M}(P) = (\text{the process that satisfies } S)$

Note that in a theorem proving environment which uses the observations of a process, we would simply use S directly as a predicate on those observations.

The graphical syntax P_G constrains the choice constructs: they are *not* general choice as is usual in TCSP. By requiring that they are labelled, we solve our problem from Section 4.1 at the expense of a loss of expression. We consider that this is a reasonable trade-off, since we are keeping intuitiveness of meaning. If the original form of general choice was required, then a user can always the normal textual form (which is still available, by design).

5 Examples

Our first example is a simple two-state process. Although our grammar does not insist on it, we would expect there to be an outermost box in most cases.



This example defines the process

$$Example1 = tick \rightarrow tock \rightarrow Example1$$

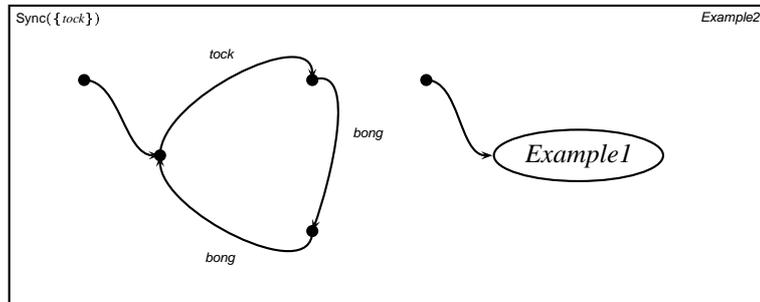
However, we might have been more comfortable defining this in terms of a predicate over untimed traces:

Example1

$$\forall tr : 1 \geq \#tick - \#tock \geq 0$$

The predicate in the box means: for all possible traces that may be observed of *Example1*, the difference between the number of *ticks* and the number of *tocks* can only be 0 or 1.

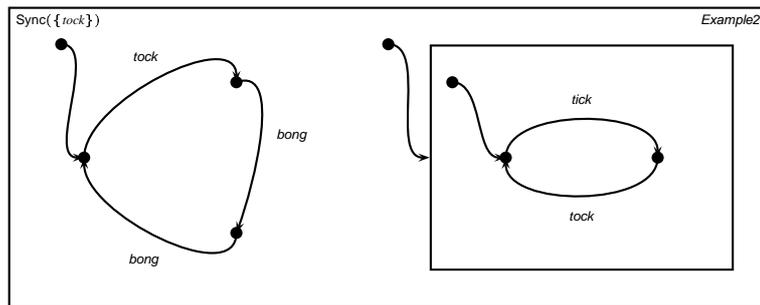
Our second example builds on the first: it describes a process that engages in two *bong* events after every *tock*.



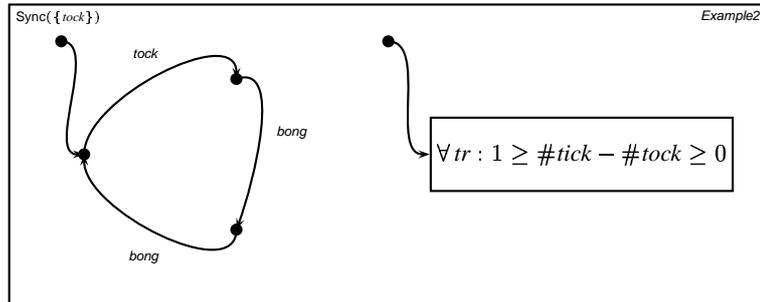
We might write this as

$$Example2 = (\mu X \bullet tock \rightarrow bong \rightarrow bong \rightarrow X) \parallel_{\{tock\}} Example1$$

Finally, we could have used *Example1* anonymously by copying it (the innermost box is not necessary; we have included it to illustrate that it can be done):



Alternatively, we could include the first example as the predicate given earlier:



6 Tool Support

The graphical notation has been designed with tool support in mind. We desire to provide for support for drawing and constructing TCSP specifications, and also for reasoning about them. Our underlying philosophy is to produce a notation (and tool support) that is compatible with and can be integrated with other existing notations and tools, e.g. TCSP, FDR, PVS, and ACL2. In particular, rather than produce our own reasoning tools, we desire to use existing tools. However, as it is impossible to predict which forms of reasoning support will be useful in all circumstances, we desire to provide tool support for graphical TCSP that makes it straightforward to combine with other tools. This approach is schematically illustrated in Fig. 1. The graphical notation, and its support tools, will be used in loose integration with reasoning tools such as FDR and PVS, as well as documentation preparation facilities such as \LaTeX .

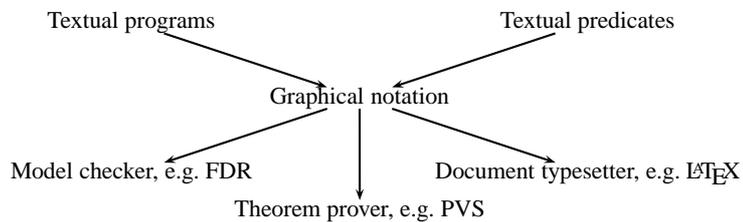


Fig. 1. Tool framework

The ongoing objectives in this line of work are as follows:

- Define a simple machine-readable language (MRL) that describes programs in the graphical notation.
- Write a drawing tool that produces programs in the MRL, and which will draw programs expressed in the MRL.
- Write a converter that takes MRL programs and transforms them into some other notation, e.g. FDR or PVS input, thus obtaining a loose integration of the drawing tool with reasoning tools.

The first and third items are mostly complete; we have a prototype tool constructed. The second item is in the design phase, but it should only be started when we are convinced that both the graphical notation and the MRL are stable and useful. The existing prototype can be viewed as a compiler: it reads in a file written in the current MRL, and then generates output code. At this time, it has a lex/yacc front end, which is used to produce a parse tree. The tree is then walked to produce output as requested by the user (i.e. PVS, FDR, or L^AT_EX output).

The current MRL is a simple declarative language, where graphical elements are given an internal name to identify them as nodes; these internal names are used as the end-points of arcs. The notation is textual, and in the case of boxes, uses BOX... ENDBOX constructs for scoping. For example, the MRL for the first two examples in the previous section are of the form:

```
i1 BOX
  NAME Example1
  i2 NODE
  i3 NODE
  INITIAL_ARC i2
  ARC i2 -> i3 ; tick
  ARC i3 -> i2 ; tock
ENDBOX
```

for *Example1* graphically;

```
i4 BOX
  NAME Example1
  i5 PREDICATE forall tr : 1 >= #(tick,tr) - #(tock,tr) >= 0
ENDBOX
```

for *Example1* as a predicate; and

```
i6 BOX
  NAME Example2
  PARALLEL {tock}
  i7 NODE
  i8 NODE
  i9 NODE
  INITIAL_ARC i7
  ARC i7 -> i8 ; tock
  ARC i8 -> i9 ; bong
  ARC i9 -> i7 ; bong
  i10 PROCESS Example1()
  INITIAL_ARC i10
ENDBOX
```

Note that we explicitly name the nodes and other entities within the graph that arcs may need to connect to. We are considering converting this to use XML; this may make future interoperability easier than with a one-off MRL.

The first theorem provers we are considering supporting in the near future is PVS. This generates an additional requirement: to generate input for PVS, we need an embedding of TCSP into PVS. We envisage building on one of the author's doctoral work [3] to achieve this. This work used PVS: for a given semantic domain, processes are defined by the observations that they admit. The meaning of the various syntactic constructs is defined by the resulting observations that may be admitted.

7 Future Work and Summary

This paper has presented a graphical notation for Timed CSP. We have given a rationale, and outlined the process by which we came to this notation. We claim that it has the major benefits of combining both graphical and textual notations in an intuitive and relatively simple fashion. We also allow textual notations to encompass both process and predicate approaches to defining processes. Additionally, we have extendible tool support for this approach as a partially-completed prototype.

We believe that this notation makes the use of formal notations easier to introduce into projects, while allowing a path towards using textual notations when proficiency with textual TCSP is achieved. Moreover, the graphical notation can stand as a useful illustrative tool, particularly for rough-sketch modelling.

Related work beyond the elements of tool construction discussed in the previous section includes

- A discussion of when it is appropriate to use each textual processes, textual predicates, and graphical notations.
- Better expressions, ideally using an identical notation to FDR.
- Other language targets for TCSP, e.g. ACL2.

References

- [1] L. Baresi and M. Pezzè. Towards Formalizing Structured Analysis. *ACM Trans. Software Engineering and Methodology* 7(1), January 1998.
- [2] G. Booch, J. Rumbaugh, and I. Jacobson. *The UML User Guide*. Addison-Wesley, 1999.
- [3] P. Brooke. A Timed Semantics for A Hierarchical Design Notation. DPhil Dissertation, Department of Computer Science, University of York; also issued as YCST 99/08.
- [4] P. Brooke, J. Jacob, and J. Armstrong. An analysis of the four-slot mechanism. In *Proceedings of the BCS-FACS Northern Formal Methods Workshop*, electronic Workshops in Computing. Springer-Verlag, 1996.
- [5] S. Brodsky, T. Clark, S. Cook, A. Evans, and S. Kent. Feasibility Study in Rearchitecting the UML as a Family of Languages using a Precise Meta-Modeling Approach. Technical Report of pUML Group, September 2000. Available at www.puml.org.
- [6] J. Davies and S. Schneider. A brief history of Timed CSP. Technical Report PRG-96, Programming Research Group University of Oxford, April 1992. Also available by FTP from [ftp.comlab.ox.ac.uk](ftp://comlab.ox.ac.uk).
- [7] J. Sun, J.S. Dong, J. Liu, and H. Wang. Z Family on the Web with their UML Photos. Technical Report TR-A1-01, School of Computing, National University of Singapore, January 2001.

- [8] Formal Systems (Europe) Ltd. Failures-Divergence Refinement: FDR 2. <http://www.formal.demon.co.uk/>, December 1995.
- [9] T. Green and R. Navarro. Programming plans, imagery and visual programming. In *Proceedings of INTERACT '95*, 1995.
- [10] T. Green and M. Petre. When visual programs are harder to read than textual programs. In G.C. van der Veer, M.J. Tauber, S. Bagnarola, and M. Antavolits, editors, *Human-Computer Interaction: Tasks and Organisation. Proceedings of ECCE6 (6th European Conference on Cognitive Ergonomics)*. CUD, 1992.
- [11] J. Grundy, R. Back, and J. von Wright. Structured Calculational Proof. *Formal Aspects of Computing* 9(5-6), 1997.
- [12] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
- [13] C. Hoare. *Communicating Sequential Processes*. Prentice-Hall International UK, 1985.
- [14] i-Logix. *Statemate: Semantics of Statecharts*.
- [15] L. Lamport. How to write a long formula. Technical Report 119, DEC SRC, December 1993.
- [16] N. Leveson, M. Heimdahl, H. Hildreth, and J. Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, September 1994. Also Technical Report 92-106 (University of California).
- [17] B. Meyer. *Object-Oriented Software Construction* (Second Edition). Prentice-Hall, 1997.
- [18] R. Paige, J. Ostroff, and P. Brooke. Principles of Modelling Language Design. *Information and Software Technology*, 42(10):665-675, June 2000.
- [19] R. Paige and J. Ostroff. Metamodelling and Conformance Checking with PVS. In *Proc. Fundamental Aspects of Software Engineering 2001*. LNCS 2029, Springer-Verlag, April 2001.
- [20] M. Petre, A. Blackwell, and T. Green. Cognitive questions in software visualisation. In J. Stasko, J. Domingue, B. Price, and M. Brown, editors, *Software Visualisation: Programming as a Multi-Media Experience*. MIT Press, 1997.
- [21] M. Petre. Why looking isn't always seeing: Readership skills and graphical programming. *Communications of the ACM*, 38(6):33–44, June 1995.
- [22] H. Simpson. Four-slot fully asynchronous communication mechanism. *IEE Proceedings*, 137 Part E(1):17–30, January 1990.
- [23] H. Simpson. Correctness analysis for class of asynchronous communication mechanisms. *IEE Proceedings*, 139 Part E(1):35–49, January 1992.
- [24] T. Systa, P. Yu, and H. Muller. Analyzing Java Software by Combining Metrics and Program Visualization. In *Proc. CSMR-2000*, IEEE Press, Feb. 2000.
- [25] M. von der Beeck. A comparison of Statecharts variants. In H. Langmaack, W.P. de Roever, and J. Vytupil, editors, *Proceedings of the 3rd International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 128–148. Springer-Verlag, 1994.
- [26] S. Owre, N. Shankar, J. Rushby, and D. Stringer-Calvert. *PVS System Guide*. Computer Science Laboratory, SRI International, September 1998.